2.5D Chiplet Architecture for Embedded Processing of High Velocity Streaming Data

by

Tomás Figliolia

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

January, 2018

Abstract

This dissertation presents an energy efficient 2.5D chiplet-based architecture for real-time probabilistic processing of high-velocity sensor data, from an autonomous real-time ubiquitous surveillance imaging system. This work addresses problems at all levels of description.

At the lowest physical level, new standard cell libraries have been developed for ultra-low voltage CMOS synthesis, as well as custom SRAM memory blocks, and mixed-signal physical true random number generators based on the perturbation of Sigma-Delta structures using random telegraph noise (RTN) in single transistor devices.

At the chip level architecture, an innovative compact buffer-less switched circuit mesh network on chip (NoC) capable of reaching very high throughput ($1.6Tbps$), finite packet delay delivery, free from packet dropping, and free from dead-locks and live-locks, was designed for this chiplet-based solution. Additionally, a second NoC connecting processors in the network, was implemented based on token-rings, allowing access to external DDR memory. Furthermore, a new clock tree distribution network,

and a wide bandwidth DRAM physical interface have been designed to address the data flow requirements within and across chiplets.

At the algorithm and representation levels, the Online Change Point Detection (CPD) algorithm has been implemented for on-line learning of background-foreground segmentation. Instead of using traditional binary representation of numbers, this architecture relies on unconventional processing of signals using a bio-inspired (spike-based) unary representation of numbers, where these numbers are represented in a stochastic stream of Bernoulli random variables. By using this representation, probabilistic algorithms can be executed in a native architecture with precision on demand, where if more accuracy is required, more computational time and power can be allocated. The SoC chiplet architecture has been extensively simulated and validated using state of the art CAD methodology, and has been submitted to fabrication in a dedicated 55nm GF CMOS technology wafer run. Experimental results from fabricated test chips in the same technology are also presented.

Primary Reader: Dr. Andreas G. Andreou

Secondary Reader: Dr. Ralph Etienne-Cummings

Third Reader: Dr. Philippe O. Pouliquen

Acknowledgments

All of the work done through all of these years wouldn't have been possible without the help and insight of many brilliant people I had the chance to meet, and share my life with. First, I want to thank my advisor Dr. Andreas G. Andreou who always believed in me, and pushed me to always think out of the box, helping me to extend the boundaries of what I would think is possible. Under his guidance, I was lucky enough to learn not only from the electrical engineering world, but from many other disciplines, which I believe today make me a better professional. His words of support through all of these years helped me to become more confident in myself.

The next person I want to thank is one of the best professionals I have ever met, Philippe Pouliquen. Every time I would be happy about an accomplishment, he would challenge me with very clever questions that would make me think about the problem from another perspective. I believe his advice and input were key to all of the good work done for the UPSIDE project. I am really thankful for all of the knowledge he was willing to share with me along all of these years.

I have come to think of the people in Andreas's lab as my second family. We

all help and support each other along the way. I really want to thank all of my lab mates, who made my stay in the lab more enjoyable. I specially want to thank Daniel Mendat who started the program the same year as me, and has always been not only a good listener, but also a very good source of advice.

I want to also thank Dr. Ralph Etienne Cummings, who's door has always been open. His words of advice have been invaluable during my studies.

Leaving Argentina to pursuit my graduate studies in the US was one of the biggest challenges I had to face. For that, I have to acknowledge my parents and siblings who always supported my decision, and on all occasions had something positive to say every time I was not feeling at my best. I also want to thank my grandparents who every year would invite my wife and I to spend some time with them at the beach, making us feel as if Argentina was not that far away. I could have never finished my Ph.D. if it hadn't been for the tireless support from my wife María Gimena, who decided to drop everything in Argentina, and join me in this adventure, and for that I will be forever thankful. I also wanted to thank my daughter to be born, Juliana, who helped me stay focused in these very difficult last months, and helped me better understand priorities in life.

Dedication

For my wife María Gimena Caíno.

Contents

Abstract **ii**

Acknowledgments **iv**

List of Tables **xii**

List of Figures **xv**

1 Introduction **1**

2 The 2.5 D nano-Abacus System on Chip and Chiplet Architecture **9**

2.1 Design Methodologies and Challenges 9

2.2 Overall Chip Architecture 12

2.3 Introduction to the CMPs' Assembly 24

2.4 Processing Units 27

2.5 Power Distribution 37

2.6 On-Chip Programmability of Clocks. 43

2.7 Stitching Logic 49

2.8 Final Layout Designs and Pinout 52

2.9 Power Up Sequence and Configuration 62

3 Clock Tree Design **71**

3.1 Clock Tree Usual Solutions 71

3.2 The Conical-Fishbone Clock Tree 74

4 NoCs **82**

4.1 First Level NoC Architecture 82

4.2 Second Level NoC Architecture 87

4.2.1 Introduction 87

4.2.2 The Proposed Network Solution 91

4.2.3 Simulation Results 105

4.2.4 Self-Diagnosis in the *L2 network* 108

4.2.5 *L2 network* Routing Tables 114

4.3 The *L1* & *L2 network* Node 124

4.3.1 Overall *L1* & *L2 network* Node Description 124

4.3.2 Input/Output Signals 134

4.3.3 Network Node Programming Capabilities 140

5 Physical Memory Interface DDR **145**

5.1 Introduction 145

5.2 *DDR DRAM PHY* Block Division 146

5.3 The *PAD_interface* Block 151

5.3.1 Delay Analysis 151

5.3.2 Operating Description 158

5.3.3 Programmable Delay Cell *SEN_DELAY* 170

5.3.4 DDR Output Generator Cell *SEN_DDR* 176

5.3.5 Input/Output Signals 177

5.4 The *PADS_alignment* Block 179

5.4.1 Operating Description 179

5.4.2 Input/Output Signals 192

5.5 The *Mux_Demux* Block 197

5.5.1 Operating Description 197

5.5.2 Input/Output Signals 200

5.6 The *Port_interface* Block 203

5.6.1 Operating Description 203

5.6.2 Input/Output Signals 211

5.7 The *Network_1_interface* Block 213

5.7.1 Operating Description 213

5.7.2 Input/Output Signals 217

6 Subthreshold CMOS Library Design 220

6.1 Introduction to Synthesis 220

6.2 Standard Cell Library Design 225

6.3 SRAM Library Design 230

6.4 SRAM Test Chip GF5 241

7 A Stochastic Architecture for the Adams/McKay Online Change Point Detection 252

7.1 Introduction 252

7.2 Algorithm Equation Development 255

7.2.1 Case of the Inverse Gamma Prior 257

7.2.2 Case of the Normal Prior 262

7.2.3 Step by Step Algorithm Computation 265

7.3 Stochastic Computing 268

7.3.1 Introduction 268

7.3.2 Stochastic Architecture for the Online CPD Algorithm 273

7.4 Stochastic CPD Test Chips GF1 & GF2 282

7.5 Stochastic CPD Test Chip GF3 290

7.5.1 Changes in GF3 290

7.5.2 Architecture Description 297

8 Design of a True Random Number Generator using RTN noise 324

8.1 Introduction 324

8.2 Closed-Loop Controlled RNG 328

8.2.1 Architectural Description 328

8.2.2 Modeling the System Noise 336

8.2.3 Considerations for this TRNG 344

8.3 Analog Sigma-Delta Random Number Generator 355

8.3.1 Architectural Description 355

8.3.2 Circuit Description 363

8.4 TRNG Test Chip GF4 377

9 Conclusions **382**

Bibliography **385**

Vita **400**

List of Tables

2.1 **Table of biases used by different PUs.** 44
2.2 **Bondpad signal assignment for all of the CMPs.** 58
2.3 **Configuring and debugging packets sent to PROG_PU.** 67

4.1 **Mean delay suffered for packets injected into a fully loaded network (random connections).** Case of a random connecting network with different number of nodes. M is the number of connections each nodes connects to. 108
4.2 **Mean throughput for packets injected into a fully loaded network (random connections).** Case of a random connecting network with different number of nodes. M is the number of connections each nodes connects to. 108
4.3 **Mean delay suffered for packets injected into a fully loaded mesh network.** This table shows the case of a mesh network with different number of nodes in the horizontal and vertical direction. . . 109
4.4 **Mean throughput for packets injected into a fully loaded mesh network.** Maximum throughput achieved in a mesh network of different number of horizontal and vertical nodes. 109
4.5 **Description of the network node interface signals.** Input and output signals found on each of the two network node versions in Figure 4.18. In parenthesis and in bold the power domain corresponding to the signal is shown. 134

5.1 **Description of the *PAD_interface* signals.** 177
5.2 **3D-DiRAM packet format.** Both read and write packet structure are presented. 182
5.3 **Description of the *PADS_alignment* signals.** 194
5.4 **Description of the *Mux_Demux* signals.** 202
5.5 **Description of the *Port_interface* signals.** 211
5.6 **Description of the *Network_1_interface* signals.** 219

6.1 **Single inverter *SEN_INV_1* propagation delay.** Delay considered for nine different voltages. For each of those voltages five corners were calculated. The last five columns help to visualize how four corners deviate from the typical corner. R stands for rising and F stands for falling. 229
6.2 **GF5 chip input pads.** Description of how input pads are shared among all of the tested blocks. 247
6.3 **GF5 chip output pads.** Description of how output pads are shared among all of the tested blocks. 248
6.4 **GF5 chip bias pads.** 249
6.5 **SRAM memory maximum clock frequency.** Maximum clock frequency measured for the four tested SRAM memory blocks in GF5 chip. 251

7.1 **Parameters in the GF1 & GF2 test chips.** List of the different parameters with the expected number of bits for each one. 286
7.2 **Choosing the maximum run-length for the stochastic CPD.** Number of false alarms, probability of hit and probability of miss for when the maximum run-length is varied. 291
7.3 **Choosing the maximum computational time for the stochastic CPD.** Number of false alarms, probability of hit and probability of miss for when the maximum computational time is varied. 292
7.4 **Maximum operating frequencies for GF5.** Different voltages supplies are used at 27C. 293
7.5 **Signals used for testing the *SRAM* blocks is GF3.** 297
7.6 **Description of the CPD block *NORM_CPD_top* interface signals.** 298
7.7 **Expected values at the input bus *bus_i* for each address in *bus_type_i*.** 301
7.8 **Comparison of block areas in GF3.** 302
7.9 **Description of the *CPD_unit* signals.** 308
7.10 **Description of the *NORM_unit* signals.** 319
7.11 **Measured clock speeds for GF3.** 320

8.1 **Standard deviation σ_P calculation.** Calculation of σ_P for $P(n)$ for the case $AL_0 = -0.8$, varying K_{dac} and K_{int}. 343
8.2 **Standard deviation σ_P calculation.** Calculation of σ_P for $P(n)$ for the case $AL_0 = -0.4$, varying K_{dac} and K_{int}. 343
8.3 **Standard deviation σ_P calculation.** Calculation of σ_P for $P(n)$ for the case $AL_0 = -0.2$, varying K_{dac} and K_{int}. 344
8.4 **Standard deviation σ_P calculation.** Calculation of σ_P for $P(n)$ for the case $AL_0 = -0.1$, varying K_{dac} and K_{int}. 344

8.5 **Areas for RNG units with different values of K_{int}.** 378
8.6 **Interface signals to the GF4 test chip.** 379

List of Figures

1.1 **Examples of wide area imagery (from[1])** . . . 3
1.2 **Processing pipeline.** . . . 4

2.1 **Input/Output Delays involved in the synthesis of a block.** Input/Output delay constraints example used in *Logical Synthesis* and *Place & Route*. . . . 13
2.2 **Overall Chiplet solution.** Diagram of the overall image processing chiplet solution. . . . 14
2.3 **Interposer picture.** Picture of the already fabricated $1\mu m$ process, $50mm$ by $64mm$ interposer. . . . 15
2.4 ***L2 network* diagram for the 128 PUs CMP.** . . . 20
2.5 ***L2 network* diagram for the 64 PUs CMP.** . . . 21
2.6 ***L1 network* diagram for the 128 PUs CMP.** . . . 22
2.7 ***L1 network* diagram for the 64 PUs CMP.** . . . 23
2.8 **Block diagram for the local bias generators.** . . . 36
2.9 **CMP1 *Yupana* PU breakdown.** . . . 37
2.10 **CMP2 *Salamis Tablet* PU breakdown.** . . . 38
2.11 **CMP3 *Soroban* PU breakdown.** . . . 38
2.12 **CMP4 *Suanpan* PU breakdown.** . . . 39
2.13 **Voltage supplies across the 128 PUs CMP.** . . . 40
2.14 **Voltage supplies across the 64 PUs CMP.** . . . 41
2.15 **C4 bumps pattern for both Network and *DDR DRAM PHY* side.** . . . 42
2.16 **Combination of C4 bumps and bondpads.** . . . 43
2.17 **Architecture for *Pulse Generator* block.** . . . 47
2.18 **Timing diagram for internal signals of the *Pulse generator* block.** . . . 48
2.19 **Block diagram for the two-input clock switcher cell.** . . . 48
2.20 **State diagram for the asynchronous circuit used in the clock switcher.** . . . 49

2.21 **Clock switcher tree used in the selection of one in nine clocks sources in the PROG_PU unit.** 50
2.22 **Timing diagram for the signals connecting the FPGA to the *L2 network*.** 53
2.23 **CMP1 *Yupana* layout ($\approx 385M$ transistors).** 54
2.24 **CMP2 *Salamis Tablet* layout ($\approx 454M$ transistors).** 55
2.25 **CMP3 *Soroban* layout ($\approx 320M$ transistors).** 56
2.26 **CMP4 *Suanpan* layout ($\approx 215M$ transistors).** 57

3.1 **H-tree and Fishbone tree architectures.** 73
3.2 **Inverted cone shape.** Shape inspiring the new *Conical-Fishbone* tree. 74
3.3 **The *Conical-Fishbone* clock tree.** Diagram of this new clock tree architecture. 76
3.4 **Implementation of the *Conical-Fishbone* tree in the CMPs.** Description of where these clock cells are present in the CMPs. 78
3.5 **Clock tree cells used in the *DDR DRAM PHY*.** Description of this augmented clock tree cell. 80
3.6 ***Conical-Fishbone* skew simulation.** Simulation results showing only up to $31.8ps$ of skew. 81

4.1 **Token ring network approach for the *L1 network*.** Dedicated network slots are assigned to each of the PUs attached to the *L1 network*. 83
4.2 ***L1 network* token-ring per row.** Architecture of the token-ring networks present for each row of PUs in each CMP. 85
4.3 **Two types of *L1 network* nodes.** Two types of network nodes were designed for the *L1 network* that allow the equidistant tapping of the PUs into the token-ring networks. 86
4.4 ***Time counter* evolution example.** Example of how the *time counter* increases when routing in the *L2 network*. 93
4.5 ***Fractional counter* update example.** This example shows how the *fractional counter* is diversified when fights arise in the routing of packets. 96
4.6 ***Fractional counter* update values.** The different values the *fractional counter* can experience according to the number of packets fighting in a node. 97
4.7 **Evolution of the distribution of packets with the same *time counter* over time.** 99
4.8 ***Fractional counter* update when trying to achieve its highest value.** 100
4.9 **Different paths arriving to the same bin.** 102
4.10 **Isolation of PUs.** Case in which the *coordinating processor* does not have access to certain PUs. 111

4.11 **The node self-diagnosing mechanism.** A simple diagram shows the mechanism by which nodes diagnose their links to other nodes. . 112
4.12 **The *SEN_SENSE* cell architecture.** Asynchronous circuit involved in the network self-diagnosis. 113
4.13 **Network address topology.** Example of the way routing could be implemented by using Cartesian coordinates as network node addresses. 115
4.14 **Combinatorial routing.** No pipelining is used in the internal routing of packets in a node. 116
4.15 ***L2 network* node.** *L2 network* network node diagram. 119
4.16 ***L2 Network* routing tables.** 120
4.17 **Routing with broken links.** Example of how the proposed way of routing performs the labyrinth strategy of "following a wall" to reach PUs almost isolated due to broken links. 123
4.18 ***Network node layout.* Description of the network node layout containing both *L1* and *L2 network* nodes.** 126
4.19 **Routing density achieved on the network node.** Over 90% of the metals' routing capability was reached. 129
4.20 **Example of usage of the network node.** 131
4.21 **Four-phase handshaking protocol.** Communication protocol between PU and its network node. 133
4.22 **Programming the PU clock.** An example is provided. 141
4.23 **Multi-slot PUs.** Example of PUs that occupy more than a single slot in the network. 142
4.24 **Network node's control words.** Different control words programming the network node. 144

5.1 ***DDR DRAM PHY* hierarchical division.** Diagram showing the different blocks composing the *DDR DRAM PHY*. 148
5.2 **Pad equalization delays.** Explanation of the three different delays involved in the equalization of the signals coming from the 3D-DiRAM. 153
5.3 ***First programmable delays.*** Example of the optimum values the *first programmable delay* should be programmed to. 154
5.4 **Plot of condition in Equation 5.4** 155
5.5 **Constraints on ΔI_d.** The admissible values for ΔI_d are the ones in the shaded region. 156
5.6 ***First programmable delay* span for different clock frequencies.** 157
5.7 **Updated constraints on ΔI_d.** The admissible values for ΔI_d, as a function of the clock period, are the ones in the shaded region. 158
5.8 **Search for the optimum *first programmable delay.*** The way in which the *first programmable delays* are tested is shown in this figure. 162
5.9 ***PAD_interface* block.** General structure for the *PAD_interface* block. 163

5.10 ***First programmable delay* + *second programmable delay* + data rate conversion.** Diagram showing the *first programmable delay*, the architecture of the *second programmable delay*, and the data rate conversion from a DDR $0.8ns$ clock period, to six SDR $2.4ns$ clock period signals. 167
5.11 **Shift-register controlling the *first programmable delay* applied to one of the 64 signals coming from the 3D-DiRAM.** Two configurations are used for this register. 169
5.12 **Synchronizer used for clock domain crossing.** 170
5.13 **Synchronizer used for clock domain crossing when data is transmitted.** 171
5.14 ***First programmable delay* architecture.** 171
5.15 **Multiplexer approaches.** Three different approaches for the multiplexer used as the minimum step delay in the *first programmable delay*. 174
5.16 **New *first programmable delay* architecture.** Architecture used for the data input *bit_DDR_i* and the negated clock *clkHn_ddr_i*. . . . 175
5.17 **Single delay architecture.** Architecture for the single delay used in the *first programmable delay* used in the data input *bit_DDR_i* and the negated clock *clkHn_DDR_i*. 175
5.18 **The *SEN_DDR* cell.** Double data rate cell. 177
5.19 ***SEN_DDR* cell timing diagram.** Timing diagram for the *SEN_DDR* cell. 180
5.20 **Deserialization of DDR bits.** Timing diagram showing the deserialization of six DDR bits into six parallel bits using a three times slower clock frequency. 183
5.21 **Pipelined Operations.** Pipelined architectures used for when inputs or outputs for an operation such as AND/OR/NAND/NOR/XOR-/MUX/DEMUX are spreaded over a very long distance. 186
5.21 **Pipelined Operations (cont.).** Pipelined architectures used for when inputs or outputs for an operation such as AND/OR/NAND-/NOR/XOR/MUX/DEMUX are spreaded over a very long distance. 187
5.22 ***PADS_alignment* block.** General structure for the *PADS_alignment* block. 189
5.23 ***Second programmable delay* training algorithm.** 193
5.24 ***Mux_Demux* block.** General structure for the *Mux_Demux* block. . 198
5.25 **Register File architecture.** Hierarchical design for the Register File used in block *Mux_Demux*. 201
5.26 **Buffers used in the communication between the *L1 network* rings through the *Network_1_interface* block and the *DDR-_DRAM_PHY*.** Architecture used for these buffers. 205
5.27 **Register file voltage domain division.** This division applies to the *Buffer NET to DDR* and *Buffer DDR to NET*. 208

5.28 **Step by step description of the functioning of the *Port_interface* block.** . . . 214
5.29 **Step by step explanation of the interface between the *L1 network* ring and the *Port_interface*.** . . . 218

6.1 **Synthesis flow diagram.** Diagram showing both *Logical Synthesis* and *Place & Route* steps in the synthesis of a design. . . . 226
6.2 **SRAM cell schematic.** Due to compactness, two SRAM cells were put together in the basic SRAM cell. . . . 231
6.3 **SRAM cell layout.** On the left the full layout of the two SRAM cells. On the right only the polysilicon and diffusion layers are shown. 232
6.4 **SRAM timing diagram.** Reset, one write operation and one read operation are performed. . . . 233
6.5 **SRAM architecture diagram.** Blocks making up the architecture of every SRAM memory. . . . 235
6.6 **SRAM current-based sense amplifier schematic.** . . . 236
6.7 **Diagram of the blocks composing the SRAM asynchronous driver.** . . . 237
6.8 **Asynchronous state diagram for signals *qw* and *qr* in Figure 6.7.** . . . 238
6.9 **Asynchronous state diagrams for both the *async1* and *async2* blocks from Figure 6.7.** . . . 239
6.10 **Asynchronous driver timing diagram.** Detailed timing diagram of all the signals in the SRAM asynchronous controller. . . . 242
6.11 **Layout for the 64x32 SRAM block.** Only up to metal three is used for all of the SRAM memories. . . . 243
6.12 **Pad sharing in GF5 chip.** Explanation of the pad sharing among the tested blocks. . . . 245
6.13 **Layout view of GF5 chip.** . . . 246

7.1 **CPD algorithm graph.** A graph explaining the time dependencies in the CPD algorithm. . . . 255
7.2 **Stochastic Encoder.** Example of numbers encoded stochastically. . 271
7.3 **Stochastic computation elements.** Example of four stochastic computational elements. . . . 272
7.4 **Stochastic architecture for the CPD algorithm.** . . . 276
7.5 **Stochastic update of the mean parameters.** . . . 278
7.6 **Bernstein polynomials block.** Architecture used in the approximation of a half Gaussian bell. . . . 281
7.7 **GF1 & GF2 layout and pinout.** Only one layout is presented as both of the chips are very similar. . . . 285
7.8 **GF1 & GF2 chip test results.** . . . 289

7.9 **GF3 chip layout with pinout.** . . . 296
7.10 **CPD cluster division in GF3.** CPD block *NORM_CPD_top* was composed of 12 clusters of 4 CPD cores each. . . . 299
7.11 **Block *NORM_CPD_cluster* is composed of four different *NORM_CPD_unit* blocks.** . . . 300
7.12 **Area comparison for chips GF1 & GF2 vs GF3.** . . . 303
7.13 **Internal division of the *NORM_CPD_unit* block.** . . . 305
7.14 ***CPD_unit* block diagram.** . . . 307
7.15 **Updated architecture for the CPD processing unit in GF3.** . 310
7.16 **Plot of functions $log(x)$ and $Mlog(x)+1$.** . . . 311
7.17 **Division based on Bernstein approximations.** Block diagram for a stochastic divider using Bernstein polynomial approximators on logarithms and exponential functions. . . . 313
7.18 **Approximation of the Gaussian bell using the Bernstein approach.** . . . 315
7.19 ***NORM_unit* block diagram.** . . . 316
7.20 **Example of the normalizing process in GF3.** . . . 317
7.21 **Conceptual block diagram for the *NORM_core* block.** . . . 317
7.22 **Video processed by the GF3 CPD chip (figure 1).** . . . 321
7.23 **Video processed by the GF3 CPD chip (figure 2).** . . . 322
7.24 **Video processed by the GF3 CPD chip (figure 3).** . . . 323

8.1 **Tradeoff between computational time and silicon area.** . . . 326
8.2 **First approach to a feedback controlled Sigma-Delta based TRNG.** . . . 331
8.3 **Second approach to a feedback controlled Sigma-Delta based TRNG.** . . . 336
8.4 **$P(n)$ standard deviation for when quantization noise is uniform.** 342
8.5 **Encoder's output.** A sample from $P(n)$ is drawn every K_{int} samples at F_{rand} frequency. . . . 345
8.6 **Scrambling architecture for the outputs of N random number generators.** . . . 349
8.7 **Auto-correlation.** Auto-correlation for the random number streams from Figure 8.3 and the result of applying the scrambling process from Figure 8.6. . . . 354
8.8 **Analog Sigma-Delta circuit model.** . . . 356
8.9 **Block diagram in the *Z-transform* domain for the Analog Sigma-Delta.** . . . 357
8.10 **Components used in the Sigma-Delta based RNG.** . . . 364
8.11 **General structure for the analog Sigma-Delta based RNG.** . 365
8.12 **Distribution for the mirrored current in the RTN transistor structure.** . . . 367

8.13 **Current distributions for *PMOS_curr* and *NMOS_curr* blocks.** Distributions for different multiplicities of transistors are shown. . . . 371
8.14 **Architecture for the comparator used in the analog Sigma-Delta.** 376
8.15 **Histogram for the comparator threshold voltage V_{ref}.** 377
8.16 **Overall diagram of the GF4 test chip.** 379
8.17 **Layout for the analog Sigma-Delta based TRNG.** Each of the major blocks belonging to this block are showed on the layout. 380
8.18 **GF4 chip layout.** 381

Chapter 1

Introduction

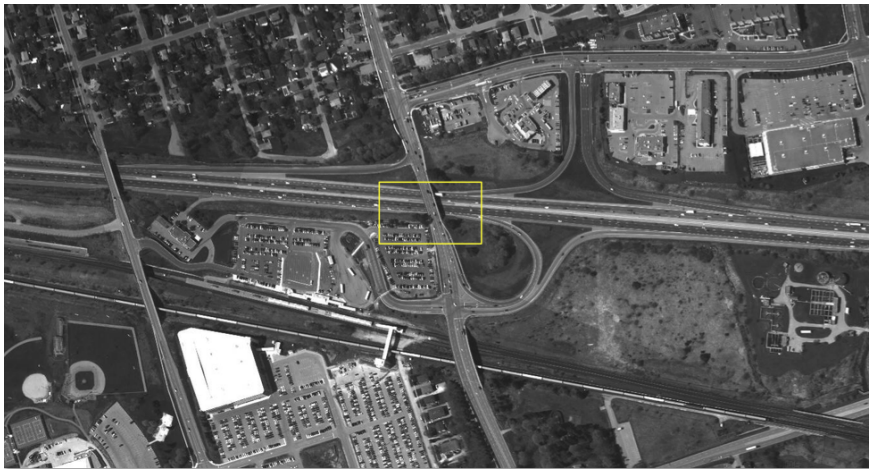
Advances in optics,[2,3] and the proliferation of cheap CMOS image sensors, have enabled the creation of commercially available larger tiled image arrays, such as the Kestrel and Simera,[4] CorvusEye 1500[5] and Sentinel CA-247,[6] with billions of pixels based on essentially what is cell-phone camera technology. Wide area motion imagery (WAMI[7]) from giga-pixel sensor systems, is a rapidly growing data resource for civilian and defense applications (see Figure 1.1). These air-borne systems, aboard a moving platform such as a small plane, a UAV or an aerostat, are capable of capturing imaging objects with an accuracy of 0.2 to 0.8 meters at a distance of a few kilometers with giga-pixel image sizes, and temporal resolution of a few frames per second[8] (3 to 15 fps). Advanced imaging technologies such as analog,[9–11] or all digital[12,13] event based cameras, can circumvent the challenges of limited frame rates, but the latter have not found their way yet into WAMI systems. Hence, WAMI processing pipelines rely

extensively on motion dynamic information.

Availability of full motion high resolution data over large, city-size, geographical areas ($\approx 100km^2$), offers unprecedented capabilities for situational awareness. The dynamic nature of the imagery offers insights about actions and patterns of activities that static images do not. Civilian applications of WAMI data, allow for the monitoring, intelligent control of traffic across large geographical areas, inference of a hierarchy of events and activities, and ultimately to recognize "life-patterns".[14] Additional applications include the coordination of activities in disaster areas, and the monitoring of wildlife. Algorithm development for WAMI tasks is facilitated through databases such as CLIF,[15] VIVID,[16] and data management standards.[1]

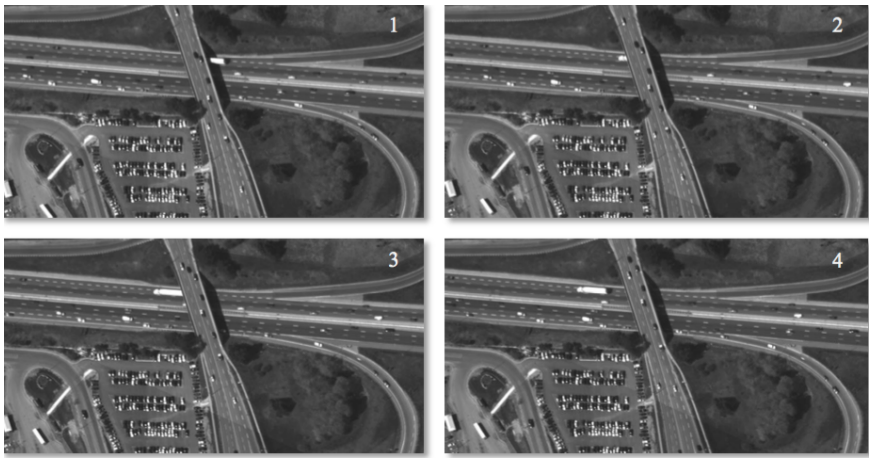
In this dissertation a system architecture for real-time *high-velocity* data processing is discussed, originating in large format tiled imaging arrays used in wide area motion imagery ubiquitous surveillance systems. A 2.5D System-on-Chip (*nano-Abacus*), implements the architecture using a silicon interposer and four application-specific chiplets. High-performance and high-throughput is achieved through approximate computing, and fixed-point arithmetic in a variable precision (6 bits to 18 bits) architecture. The architecture implements a variety of processing algorithms classes, ranging from convolutional networks (ConvNets), to linear and non-linear morphological processing, probabilistic inference using exact and approximate Bayesian methods, and ConvNet based classification.

A reconfigurable computing based, processing pipeline architecture (Figure 1.2)



(a)

(b)



(c)

Figure 1.1: Examples of wide area imagery (from[1])

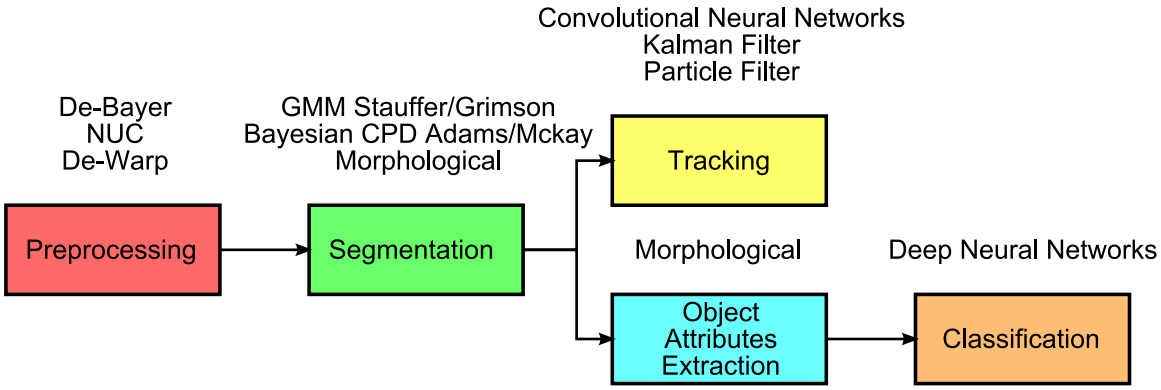


Figure 1.2: Processing pipeline.

was developed to emulate the computational structures for a System-on-Chip (SoC) that will be fabricated in the 55nm GF CMOS technology. Mapping the algorithms on a reconfigurable computing platform has a dual goal: (i) algorithm exploration and (ii) architecture exploration. The processing flow begins with raw pixel values from a camera, implementing De-Bayering interpolation, non-uniformity correction, camera motion compensation, background/foreground segmentation, object attributes extraction, object tracking, and object classification. This processing pipeline was implemented entirely using event based neuromorphic and stochastic computational primitives. This FPGA-based designed system, upon which *nano-Abacus* was based, was capable of processing in real-time 160 x 120 raw pixel data running on a reconfigurable computing platform (5 Xilinx Kintex-7 FPGAs).

The overall design philosophy of *nano-Abacus* System-on-Chip, as well as its final assembly and choice of the different available processing units is presented in Chapter 2. The 2.5D *nano-Abacus* SoC comprises of a $50mmx64mm$ silicon interposer (5 metal layers, four stitched reticles) and five "chiplets". Two of the chiplets are COTS

components; a *Xilinx Zynq-7100* die for operating system support and high speed I/O, and a high-bandwidth memory stack (*Tezzaron GEN4 3D DiRAM*). Three additional heterogeneous chip multiprocessor (CMPs) "chiplets" are chosen from a pool of four, designed in 55nm GF CMOS, implementing mixed-signal programmable and reconfigurable processors for energy efficient, high velocity Big-Data computing from motion area imagery. The *nano-Abacus* SoC is not an ASIC, but rather a processor architecture that can be hardware or software re-configured, with three of four "chiplet" processor designs, all with common physical standard footprint and logical interfaces. The *nano-Abacus* chiplet-core consists of a high bandwidth memory interface (*DDR DRAM PHY*), a Level-1 token ring network on chip (*L1 network*) allowing each processing unit to have access to the external DDR memory, a Level-2 switched circuit mesh network on chip (*L2 network*) for the communication among processors, and a general purpose input/output port (GPIO).

The design of a new clock tree architecture (the Conical-Fishbone tree) well suited for the operational requirements of the *nano-Abacus* chiplets is presented in Chapter 3. The design ensures that the impedance seen at the output of each of the tree active drivers is exactly the same on each clock tree level, and it is inspired by the shape of an inverted cone. Driven by the clock at the tip of the inverted cone, subsequent levels of the clock tree hierarchy are driven in a geometrical progression. If several cross sections are created in the inverted cone, and one considers each of these resulting rings to be one of the many nets in a Fishbone clock tree, one can observe that if

a ring is excited evenly from the ring below, the circular characteristics of the wire will make the effect of reflections be exactly the same along any place in the wire. Simulated outputs of a $1.25GHz$ clock propagated through the clock cell in the *DDR DRAM PHY* shows a maximum skew of $31.8ps$ for clock outputs spreaded evenly for almost 14mm long. These simulations have been done considering all the extracted parasitics from the clock tree layout.

The architectures for the first and second level NoCs are discussed in Chapter 4, a Level-1 token-ring network on chip (*L1 network*), a Level-2 switched circuit mesh network on chip (*L2 network*). While the design of the physical layout and performance of the *L1 network* has been challenging, the *L2 network* necessitated an innovative design approach characterized by the requirements of no packet loss, minimal usage of resources (reduction in silicon area), and a minimal finite latency for a packet to get delivered, warrantying no dead-locks or infinite loops (live-locks) for packets. The results of the theoretical analysis, as well as extensive simulations, are presented, demonstrating that the *L2 network* is capable of meeting the operational requirements while operating in excess of $300MHz$, with a theoretical maximum total throughput of $9.8Tbps$, and a $1.6Tbps$ total throughput for the implementation in these chiplets.

Chapter 5 presents the design of the *nano-Abacus* chiplet high performance PHY interface. The *DDR DRAM PHY* IP interface is a high speed mixed-signal design task with strict physical layout and placement constraints. Through an innovative

and physical design methodology, a simulated throughput of 2.5*Gbps* per bit line, per direction, is measured. The connection to each of the hosts is done through a bidirectional 64-bit DDR interface, allowing for a maximum bidirectional throughput of 320*Gbps*. A detailed *System Verilog* model of the memory was supplied by the memory vendor, allowing to perform close to full-chip logical simulations. Creating an interface to this external memory posed several challenges that had to be addressed, such as the design of custom architectures for programmable delay lines used in the equalization of every bit line coming from the memory, the design of algorithms in performing delay training, the custom design of clock tree cells.

Key ultra-low voltage library components are designed, simulated, fabricated and tested, and results are presented in Chapter 6. More specifically, analog and SRAM cells that can operate at subthreshold voltages (as low as 400mV). For the ULV SRAM, an asynchronous driven architecture capable of operating at 400mV power supply has been fabricated and tested successfully. This block SRAM component has been incorporated in a fully customized CMOS standard cell library. Additional work employed CAD tools to fully characterize the behavior and geometries of each cells in the standard cell library when operated in sub-threshold CMOS. At a power supply of 600mV, the performance of the SRAM blocks has been measured to $374MHz$ for the 64 words cell, while the largest block (512 words) is measured operational at $136MHz$.

In Chapter 7 an event-based stochastic architecture for the Adams/McKay Bayesian

Online Change Point Detection algorithm (BOCPD[48]) is reported. Change point analysis (CPA), also known as change point detection (CPD), is the identification of sudden and often small changes to the statistical parameters or the output of a system that is in the form of sequential data. Often CPA is employed for the segmentation of a signal to facilitate the process of tracking, identification or recognition. Here the algorithm by Adams/McKay for online Change Point Detection is used. The architecture employs probabilistic event representation and computational structures that natively operate on probabilities. A fully programmable multicore CPD processor was synthesized in VHDL. This first architecture approach is capable of processing in real-time 160 x 120 raw pixel data running on a single Kintex 7 FPGA (Opal Kelly XEM7350-K410T). The architecture was also, fabricated in three 55nm CMOS test chips. Experimental results from the test chips are presented in this chapter.

The system architecture for a Bernoulli random number generator with a true probability $p = 0.5$, is the subject of Chapter 8. The architecture is based on the perturbation of an analog Sigma-Delta modulator using random telegraph noise (RTN) from a single MOS transistor. The architecture involves multiple feedback control loops analyzed and simulated to assure stability. A test chip that incorporates 144 units was designed and fabricated in the 55nm CMOS technology. Each RNG unit occupies $\approx 2200\mu m^2$, and it was simulated to operate from $1MHz$ to $25MHz$ while consuming $\approx 432nW$ per MHz.

Chapter 2

The 2.5 D nano-Abacus System on Chip and Chiplet Architecture

2.1 Design Methodologies and Challenges

When designing micro-chips, different approaches can be taken. One can perform *logical synthesis* and *Place & Route* in a flat fashion, or a more modular bottom up approach could be taken. The term *logical synthesis* will be given to the translation of hardware description written in *VHDL* or *Verilog*, into a netlist of logical cells belonging to a particular standard cell library. On the other hand *Place & Route* tools will take that translation from the *logical synthesis*, and will perform the actual physical implementation of that netlist, laying down the actual layout for every single cell and performing the corresponding metal interconnections. For small silicon areas,

usually flat *Place & Route* results in a more efficient outcome regarding power, area and timing, mainly because the used *Place & Route* tools are provided with all the degrees of freedom for the considered design. When bottom up approaches are used, at every level of hierarchy considered, the degrees of freedom are reduced, and then the result obtained might not be optimum. In the flat *Place & Route*, logic that might be repeated several times in different modules, could be just collapsed into a single unit, resulting in area reduction and less power dissipation. So, when would a bottom up approach be the answer? As technology advances, chip areas are increased, and feature sizes decreased, resulting in very large relative silicon areas, for which the complexity and time used performing *Place & Route* increases exponentially. Sometimes the time span used in performing Place & Route for very large designs could be weeks, and it is just with a simple modification to the design, that this process needs to be restarted. It is for this reason that modular designs are becoming more attractive, allowing to keep complexity limited at every level of hierarchy.

Bottom up approaches face challenges that flat designs do not. A wider understanding of the involved physical design is required. Power distribution and timing becomes more challenging as these aspects need to be analyzed individually for each block, and eventually their impact in an upper level of hierarchy. Modular designs additionally necessitate the specification of additional constraints, such as the timing requirements in the signals connecting each block to a top level. Dimensions need to be specified for each block. Shape and size for hierarchical blocks need to be care-

fully chosen when the broader picture is considered. The position of pins on each block becomes an important issue. A poor choice for their position might result in the passing of timing constraints local to the block, but the violation of them when timing is analyzed in the upper level of hierarchy. Due to the complexity of the designs presented in this work, modular design is a must. Up to six levels of hierarchy were used in some of the blocks that compose the processing architecture that will be proposed, making proper physical planning necessary. Fortunately, the choice of this physical planning path, allowed to reduce the time performing *Place & Route* for the top level of all of the chips designed in this project to under a day.

Concepts such as tiling, block abutment and module repetition start showing up in large designs. In the face of tight timing constraints, abutment of blocks becomes a necessity, and with it, proper input/output delays need to be correctly equalized as well as the position of the pins connecting all of the abutting blocks. When having two blocks utilizing the same clock signal, one block might send data to the other block, and the later block might send data back as well. Let's consider the case of block A sending a bit of information to block B. The *input delay* constraint for block B is the time the bit at the output of block A takes to travel from the closest register in block A to the physical block pin. On the other hand the *output delay* for block A is the time that the before-mentioned bit takes to travel from the input pin in block B to the closest register's input still in block B. These timings are shown in Figure 2.1. If the summation of the input and output delays in the abutting blocks

is not less than the desired clock period, then no optimization in the upper level of hierarchy will ever allow that clock frequency to be achievable. It is for this reason that the choice of input and output delays in a block is of the highest importance. In the design proposed, the existence of several processing units (PUs) will be shown. These units will need to abut with each other to create a compact overall design, where analysis of these timing constraints need to take place. When performing this abutment, additional safe measures will need to take place when designing these units, such as *Place & Route* blockages that will prevent design rule violations, or cross-talk between internal block signals at the place where blocks abut.

An additional concept used in large designs is module repetition. Repetition of modules not only allow to have a more consistent timing and power distribution over the chip, but it also simplifies the synthesis of the whole chip. This is the case of the two networks on chip (NoC) that have been implemented in the design of the chips presented in this work. For these networks, each node is a block that is repeated all over.

2.2 Overall Chip Architecture

In this work, the design of four large chips is presented (17.466mm by 14.133mm) designed in $55nm$ GF process. Three of the four designed chip multiprocessors (CMPs) will be mounted on an interposer chip ($1\mu m$ process of size 50mm by 64mm),

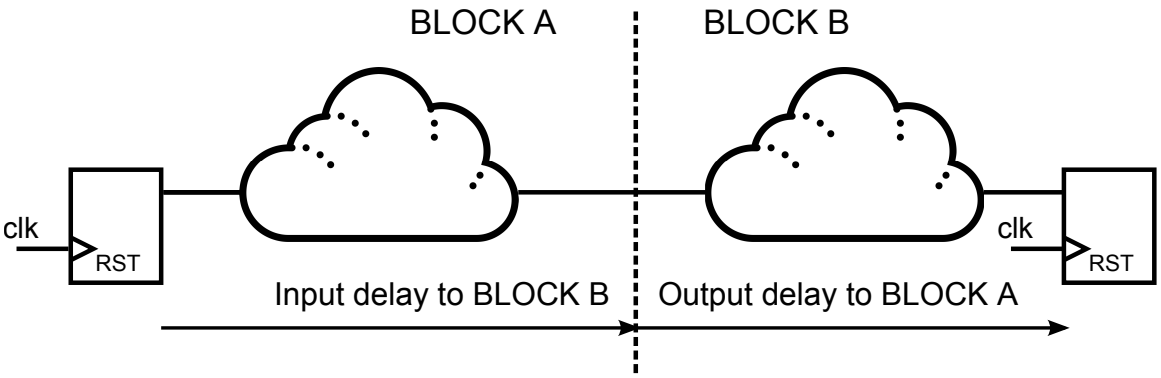


Figure 2.1: Input/Output Delays in a synthesized block. Two different delays need to be taken into account when connecting two blocks that are abutting. The summation of both delays need to be less than the desired clock period.

along with a state-of-the-art package-less FPGA (Xilinx Zynq 7100 FPGA) and a 3D-DiRAM memory (provided by *Tezzaron Semiconductor*). Several options were considered for interconnecting all the chips, but an interposer chip resulted to be the best one when power needs to be reduced as much as possible. The main advantage in building an interposer for connecting all of the units, is that the capacitance of the lines in the interposer is much less than in a PCB, and additionally the overall design achieved is much more compact. This is depicted in Figure 2.2. Each of the four chips multiprocessor will perform processing on images of up to 400Mpixels through the usage of massively parallel processors. All of the CMPs will have access to an on-interposer 3D-DiRAM memory stack and additionally will communicate with an on-interposer FPGA. Each of these CMPs is composed of either 64 or 128 processing units (PUs). With the selection of different types of PUs on each chip, different image-processing flows can be achieved, and it is for this reason that the choice of these different PUs is desired to be something that can be easily changed without

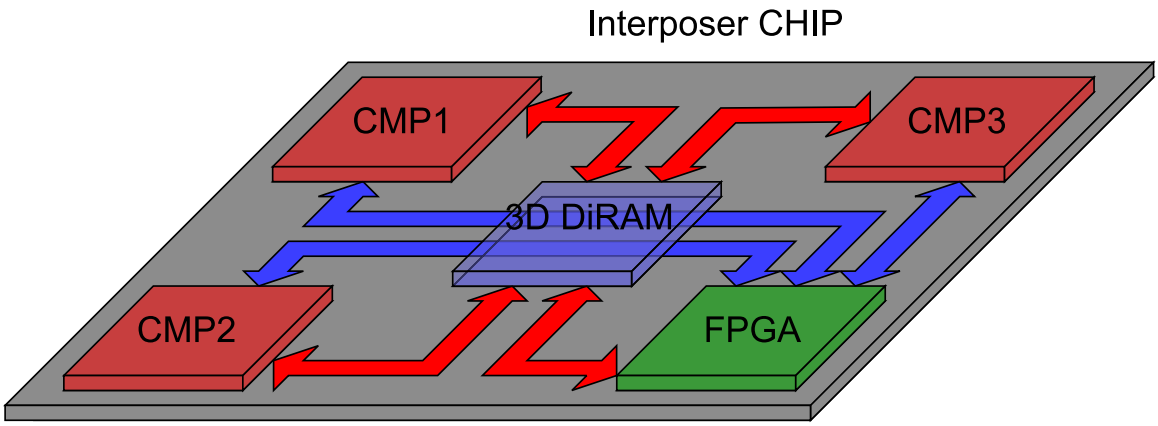


Figure 2.2: Overall Chiplet solution. In this picture the overall result of the UPSIDE project is presented, with the interconnection of three of the four CMPs, an FPGA and a 3D-DiRAM stack. All of these chips will connect through an already fabricated interposer in $1\mu m$ process with a size of $50mm$ by $64mm$.

having to start the design of each chip all over again. The different types of PUs will be addressed in one of the lasts chapters, because there is no need in knowing what the PUs will do at this point. The design of the interposer was done by Philippe Pouliquen and Gaspar Tognetti. A picture of the interposer is shown in 2.3.

When designing large chips, problems such as clock tree integrity, exponentially increasing time for synthesizing, placing and routing designs, and difficulty in performing minor changes, become more problematic. With very long distances for a clock signal to travel, mismatch and variations along the die will make it very difficult for a clock tree to achieve the desired skew, slew and speed. Other options such as building H-trees with very strong drivers will work, but most likely a modular design will find this alternative very difficult to deal with, because of the specific places the clock drivers need to be placed. The amount of time in performing *Place & Route* on a design, for tools such as *Cadence Innovus*, increases exponentially with

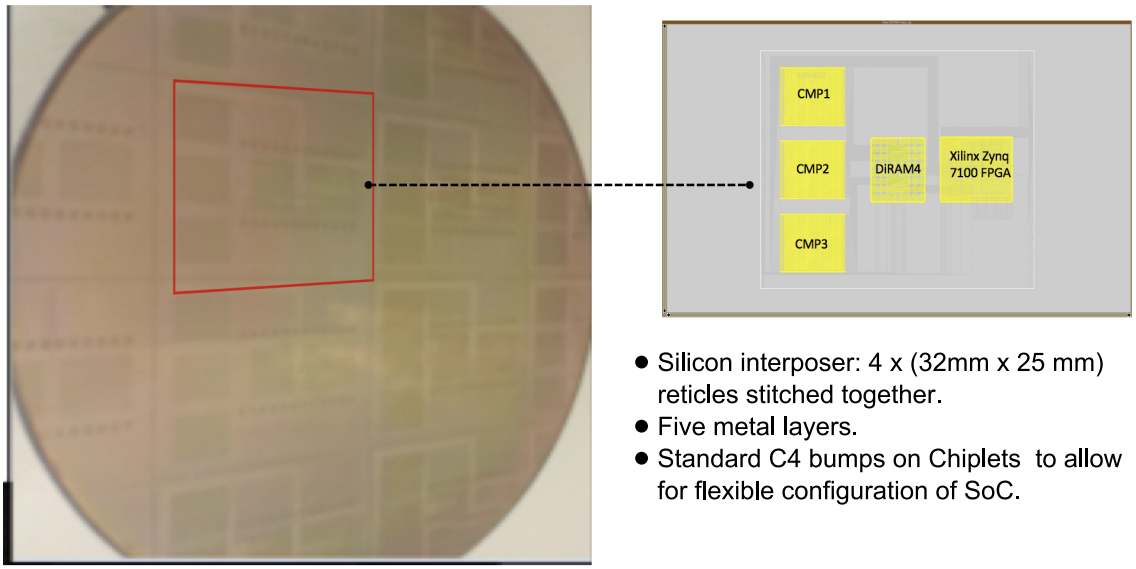


Figure 2.3: Interposer picture. Picture of the fabricated interposer. Four $32mm$ by $25mm$ reticles stiched together, with 5 metal layers and standard C4 pads.

the size of that design. Consequently, if modularity is not exploited enough, one can be found in a situation where several days are required to perform *logical synthesis* and *Place & Route* because of just a minor change to the design. If on the other hand one can exploit as much as possible the bottom up approach synthesis, taking advantage of repetition in the design, the time spent could be reduced significantly. All of the CMPs were designed in this way, breaking the design into smaller and more approachable problems. The only disadvantage to this approach is that one has to have a very clear picture of the full chip layout, especially when talking about power planning.

In the designed chips modularity was exploited as much as possible, with the main objective of easing the task of assembling the final four chip designs. If no consideration is given to the content of each of the processing units on each chip,

only two main designs arise, the one with 64 PUs and the one with 128 PUs. The design with 64 PUs is composed of eight rows of eight PUs each. The one with 128 PUs is composed of eight rows of 16 PUs each. The whole point in building modular chips is that major changes can be applied to it without spending any additional time redesigning anything. The functionality of a chip can be completely changed in just a few minutes by replacing PUs by other ones. But in order to do so proper planning needs to take place.

For the communication in between PUs a buffer-less mesh network was designed. This network will be called the *L2 network* (L2 stands for level two), and it has very particular and convenient characteristics that will be addressed in a later chapter. An additional network, called *L1 network*, was designed, allowing each PU to have access to an external 3D DDR memory. If a standard interface can be designed between the PU and its local node connecting to the *L1 network* and *L2 network*, an interface that does not rely on any particular clock (asynchronous interface), then the two top level designs for the 64 PU and 128 PU CMPs can be completely abstracted from the content of the PUs. Each of the PUs will have its own clock tree completely independent from any other clock in the system. This is the reason why a four-phase handshaking protocol interface was designed for the communication of each PU with both *L1 and L2 networks*. This allows placing "dummy" PUs for both main designs, and later replace them with the final desired PUs. The *L2 network* can be seen in Figure 2.4 and 2.5. The connection to the FPGA for both 64 and 128 PUs designs

is done through different network nodes, $(1, 0)$ and $(1, 3)$ respectively, because of physical constraints that will be later mentioned. The communication between the FPGA and its network node uses the same protocol any of the PUs uses with its own node, with the difference that serializers and deserializers needed to be used for the FPGA due to the extremely wide network bus, which is over 300 bits. Additionally, so that throughput to and from the FPGA could be increased, bidirectional pads were used for the communication in between the *L2 network* and the FPGA. These two figures show the different addresses used for each node when performing destination based routing on chip.

Access to DDR memory is granted to each of the PUs by incorporating the additional *L1 network*. This network will communicate each of the PUs with the DDR memory through a high speed memory interface called *DDR DRAM PHY*. This block will translate read and write requests from the PUs to the 3D-DiRAM memory. This network is formed by independent token-ring networks on each of the rows in both 64 and 128 PU CMP designs. Each of these token-ring networks will have a dedicated *DDR DRAM PHY* port. A total of eight different token-ring networks will be present for this *L1 network*, and each PU will communicate with its *L1 network* again through a four-phase handshake protocol interface. The *L1 network* can be seen in Figure 2.6 and 2.7.

When these CMP chips need to communicate to the outside world, it can either be done through the DDR memory or through the *L2 network* connecting to the

on-interposer FPGA (see Figure 2.2). The *L2 network* will have an additional node, apart from the 64 and 128 previously mentioned nodes. This node will have only access to the *L2 network*, and the processing unit assigned to this node will actually be an external FPGA to which the *L2 network* will connect through the left side pads of the chip. The FPGA will be able to send and receive packets to and from the *L2 network* using the four-phase handshaking protocol, and will also have its own address, making the communication between FPGA and PUs completely transparent. The utilization of this asynchronous protocol in communicating the FPGA with the *L2 network* is very convenient as it does not require the equalization of any of the data bits lines with respect to a received clock.

Each of the CMP chips is $17466\mu m$ by $14133\mu m$ in size. Because of these large dimensions, as mentioned before, it is impossible to expect the *Place & Route* tool to create clock trees with very low skew and slew. It is for this reason that a custom architecture was designed for the clock trees in the design. Long clock tree cells of size $\approx 1500\mu m$ by $\approx 50\mu m$ were designed. These cells would take a clock input and will generate several clock outputs along one or both long sides, with a skew of only up to $30ps$, allowing clock speeds of up to 1.25GHz to be propagated through these cells. These cells allow clock trees to be built local to the outputs of these clock tree cells, making these clock trees much smaller and more reliable. In Figure 2.4, 2.5, 2.6 and 2.7 the different clock cells that allow both networks to be completely in sync can be seen. Similar cells were used for distributing an asynchronous reset to the

network.

A highly conductive power grid was designed on top of all the chips. These grids hold the different power supplies for different voltage domains across the chip and additionally supply external and locally generated biases that are made available to all the processing units. The locally generated biases are provided by a local *Band Gap Reference* that will be placed in one of the PUs. The total number of external biases is 16, and 16 is also the total number of locally generated biases. These power grids were designed in metal 7 and 8, allowing the design of each processing unit to span from metal 1 to metal 6. If a power supply or bias is locally required in a processing unit, a simple connection to metal 7 can be generated. It is for this reason that it was decided that providing a template with the exact position of the power grid and standard pins connecting the PU to both networks was the way to go for everybody designing PUs. This template would ensure compatibility when placing or replacing PUs in the network. Additionally, the clock provided to the PU from the network node is a programmable one, so if more than one PU slot was needed for a particular design, as long as the different clocks provided to each of the PU slots are configured to have the same frequency, these clocks would actually match also in phase. This characteristic would allow local clock trees to have more than one root, making local trees have a reduction in their depth, allowing then a better reliability. This means that, the person designing that multi-slot PU can rely on several clock inputs that are in phase, reducing the complexity of the local clock trees.

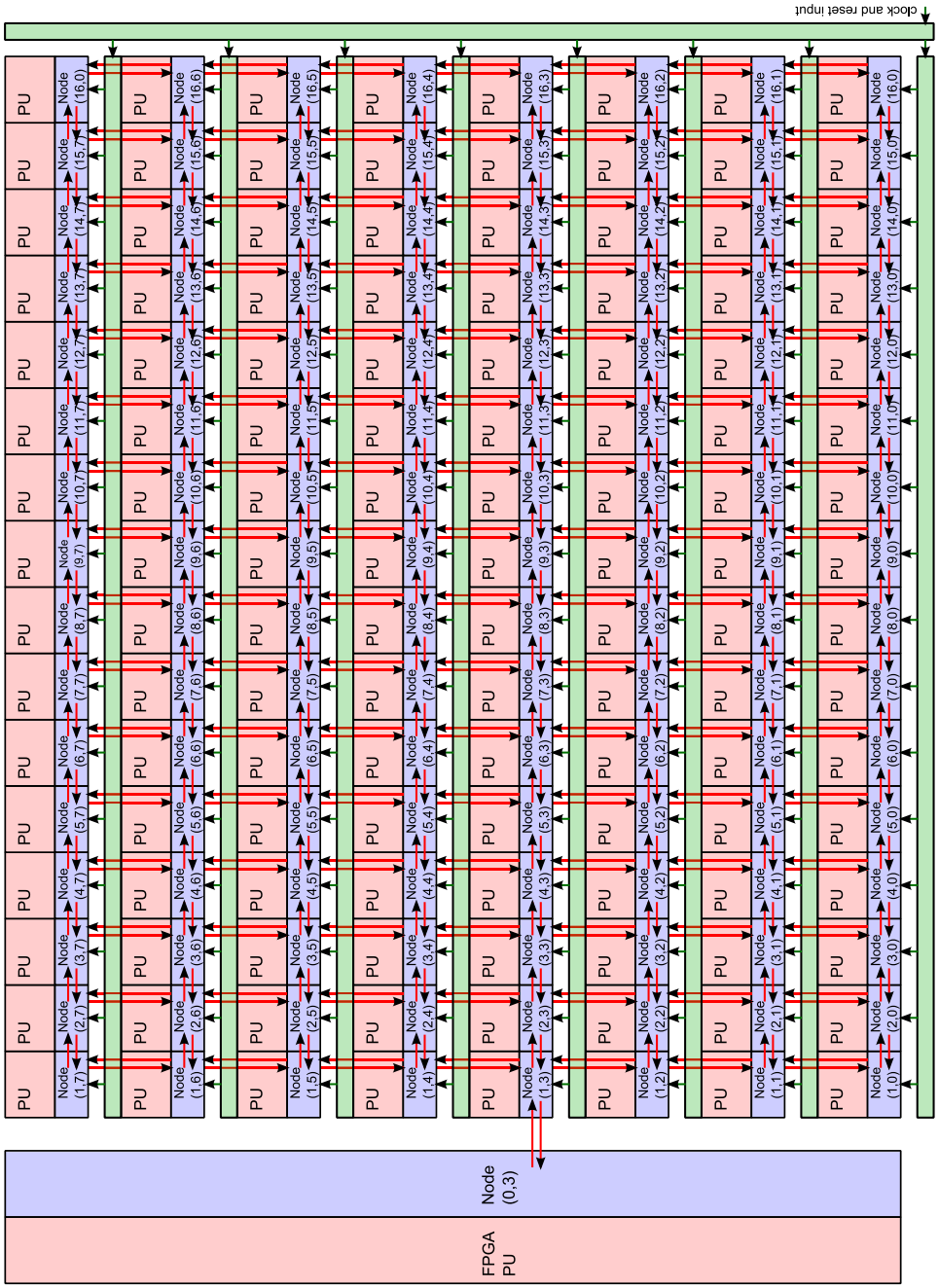


Figure 2.4: *L2 network* for the 128 PUs chip. Communication to the FPGA is done through the $(1,3)$ node. The communication between node $(0,3)$ and the FPGA is done through bidirectional pads placed on the left of the chip. Each of the packets in the network contains 256 bits of data, making it really difficult to have that same number of pads in that communication. A serializer and deserializer are being used to send and receive packets between the *N2 network* and the FPGA. The green blocks distribute reset and clock signals.

Figure 2.5: *L2 network* for the 64 PUs chip. Communication to the FPGA is done through the (1,0) node instead of (1,3). The 64 PUs chip is shared with UCSB (University of California at Santa Barbara), and then some of the chip area was assigned to them on the left side of the chip, making it more complicated to perform the FPGA connection through a node addressed (1,3). Each of the packets in the network contains 256 bits of data, making it really difficult to have that same number of pads in that communication. A serializer and deserializer are being used to send and receive packet between the *N2 network* and the FPGA. The green blocks distribute reset and clock signals.

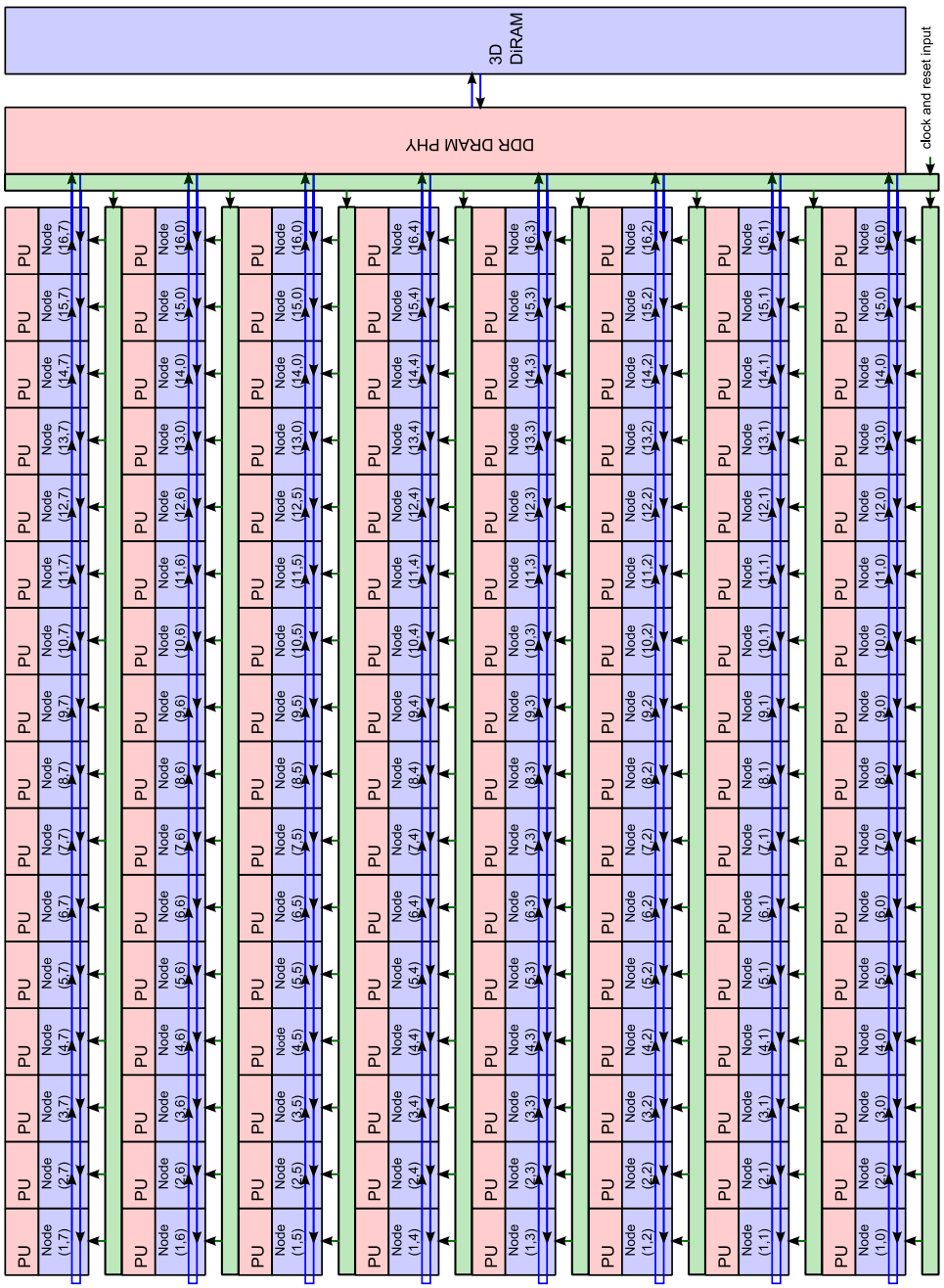


Figure 2.6: *L1 network* for the 128 PUs chip. Eight different token-ring networks communicate with the *DDR DRAM PHY*. The communication between the *DDR DRAM PHY* and the DDR memory is done through two DDR buses, where each bus is composed of 66 signals, 64 data signals and two complementary clocks. The required pads communicating with the DDR memory are placed on the right side of the chips.

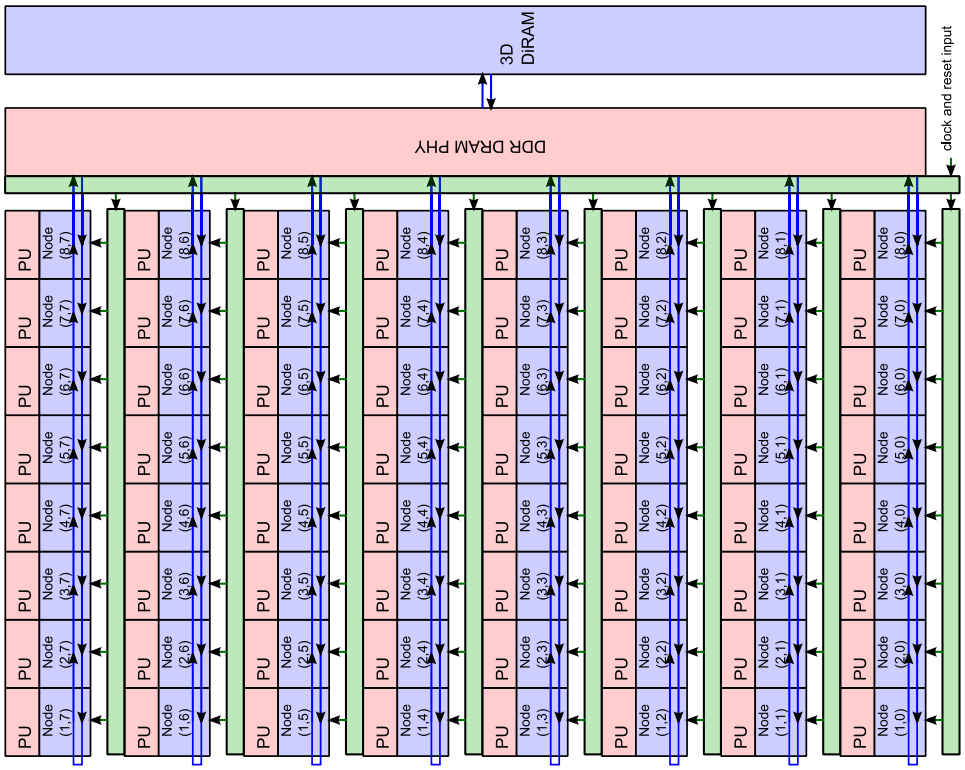


Figure 2.7: *L1 network* for the 64 PUs chip. Eight different token-ring networks communicate with the *DDR DRAM PHY*. The communication between the *DDR DRAM PHY* and the DDR memory is done through two DDR buses, where each bus is composed of 66 signals, 64 data signals and two complementary clocks. The required pads communicating with the DDR memory are placed on the right side of the chips.

2.3 Introduction to the CMPs' Assembly

The general architecture for the four designed CMPs was presented. Three of the CMPs would contain 128 PUs, and the forth one would consist of 64 PUs. The reason why the last CMP has only 64 PUs is that part of the area was designated to UCSB (University of California at Santa Barbara).

Two very distinct NoC architectures were implemented in these chips. A *L1 network* would provide communication to an external 3D DDR memory through a custom designed high-speed memory interface, featuring local delay training for each line coming from the memory . With the help of a token ring network on each of the CMP PU's rows, and with an arbitration scheme among rows consisting of an additional token ring network implemented internal to the high-speed memory interface (see Chapter 5), every single PU is able to write and read to and from external memory. With the usage of a $1.25GHz$ clock, speeds of up to $160Gbps$ can be reached in the path going and coming from the 3D DiRAM, for a total of $320Gbps$ throughput. Considering that for both write request and read answer the packet size is 384 bits, 256 of those bits actually correspond to data. Taking this into account, one can calculate the effective total throughput achieved is $213.3Gbps$.

For the second NoC, a novel way of performing buffer-less routing was implemented in the communication among all of the PUs in a single CMP. This network is free from dropping packets and it has been demonstrated not to have dead-locks or live-locks under any condition. The *L1 network* features a word size of 256 bits, and then the

same word size was selected for the *L2 network* as it made it easier to forward packets from one network to the other.

Simulations were run where every PU would send packets to random destinations. As it will be shown in Chapter 4, as long as a free link is found by a PU, then a packet can be injected to the *L2 network*. In these simulations, for every free link found by a PU, a packet is injected. This would saturate the network, filling all of the possible links connecting all of the *L2 network* nodes. In this scenario throughput of up to $9.8Tbps$ were theorized for the 128 PU CMPs, running at $300MHz$ network clock. When considering that an asynchronous four-phase handshaking protocol has been used in the communication of every PU with its network node, the simulated throughput was reduced to $1.6Tbps$. The clock provided to each PU is programmable, being generated from the one provided to the NoCs. The throughputs mentioned are extracted considering the case in which the PUs use the same clock frequency as the NoCs.

From the CMP point of view, communication to the outside world can be done in two ways. The first one is through external memory, which is shared by three CMPs and an FPGA on the custom designed interposer. The second one is directly through the *L2 network*. On the left side of the CMPs, a bus communicating the FPGA to the *L2 network* was implemented. This bus can be thought as the communicating medium between a PU, which in this case is the FPGA, and its *L2 network* node. Because of the limited number of pads, a serializer and deserializer had to be implemented

in its communicating protocol. Because of the fact that a four-phase handshaking protocol is still used in this interface, equalization for all of the interposer wires was not necessary in comparison with the ones connecting to the 3D DiRAM as seen in Chapter 5.

Power domains with different voltages across the chips were implemented with the objective of reducing power consumption. Additionally, the usage of a low voltage standard cell library allowed all of these voltages to be tunable for values down to $400mV$ (see Chapter 6). A high conductivity grid was implemented for the distribution of all of the supply voltages, as well as 32 distinct bias network grids capable of being used by any mixed-signal PU. With C4 bumps, all of the voltage supplies are distributed across the chip, as well as all of the logical signals connecting to the interposer. Additional to C4 bumps, bondpads were added in the perimeter of the chips, allowing for a full functional standalone test-bench without the necessity of the interposer. Not only logical signals were replicated with bondpads, but all of the voltage supplies and biases as well.

A novel architecture for performing background-foreground segmentation is presented in Chapter 7, based on the ChangePoint Detection algorithm.[48] This architecture would provide a very compact design that allows for a massive parallelism in analyzing high resolution images. Additionally, a true random number generator based on a RTN perturbed Sigma-Delta converter was designed (see Chapter 8), allowing to generate all of those random numbers required by the CPD architecture.

Unfortunately, time was not enough to port the design into a PU capable of being placed in the CMPs, but fortunately students at Andreas Andreou's lab helped in the design of several types of PUs. Very diverse ways of processing were introduced with these units, where some of them would feature charge based primitives, other ones would rely on non-volatile memory (NVM) or they would be just purely digital, for example.

2.4 Processing Units

Four CMPs were assembled using the processing units that are described here:

1. **M0_PU** & **dual_M0_PU** (designed by Alejandro Pasciaroni) (5 PU slots, $\approx 12.7M$ transistors and 10 PU slots, $\approx 25.3M$ transistors respectively).

   This PU implements the CORTEX M0 processor, and an AMBA bus. In the integration of this processor to the CMPs, peripherals have been attached to the AMBA bus, augmenting the processor's capabilities. Due to the considerable latency to external memory, a memory hierarchy has been locally implemented. This hierarchy consists of a cache memory attached to the AMBA bus, serving as an L1 level memory. An additional tightly coupled memory is connected to the cache, serving as L2 level memory. The system additionally implements a direct memory access unit (DMA), allowing memory transfers between the main external memory and L2 memory. These transfers do not interfere with

the processor's operation since the DMA can operate in parallel with the processor as long as both access different memory locations. Finally, a quad-SPI receiver/transceiver has been also incorporated to the system.

Two versions of this unit were designed, one of them containing a single M0 processor, and the second one featuring two. Due to lack of space in the 64 PU CMP, this chip was assigned the PU containing only one processor. In addition to the bidirectional bus used in the communication of a PU with its network node, two more interfaces were added to this unit, allowing communication and booting of the processors without having to go through the *L2 network* connection to the FPGA. The first interface is UART, using four wires to communicate with each of the M0 processors. The second interface is a quad-SPI one, using ten wires in the communication to only one of the M0 processors for both PU versions. The signals corresponding to these two interfaces were assigned to pads of the left side of the chip.

2. **8LMO_PU** (designed by Martin Villemur) (2 PU slots, $\approx 5.2M$ transistors).

   This design implements the Piece-Wise-Linear algorithm[83] for linear functions. Vector-vector multiplications can be performed for up to 4096 words of 8-bits each.

3. **1MO_PU** (designed by Martin Villemur) (2 PU slots, $\approx 5.8M$ transistors).

   This is a squared bi-dimensional array of processors. The processing is done on

single bit data inputs, with 3-bit local states and local squared inter-connectivity with unitary distance. This unit is thought for the morphological processing of binary images utilizing first order statistical filters. This array is coordinated by a custom *Harvard* based processor with 13 instructions of 8-bits each. This processor features three pipelined stages, and an interface based on *ARM*'s AHB-Lite. A 8Kb local memory was implemented.

4. **8NMO_PU** (designed by Martin Villemur) (1 PU slot, $\approx 3.0M$ transistors).

   This is eight Simplicial Cellular Neural Network (SCNN)[84] linear arrays, where each of them features 64 Piece-Wise-Linear (PWL) processors modified for the calculation of symmetric functions. The eight arrays can be configured to work simultaneously as a single 512 processors array, or 8 single ones in a pipelined fashion.

5. **IFAT_PU** (designed by Jamal Molin) (2 PU slots, $\approx 3.2M$ transistors).

   The Integrate-and-Fire Array Transceiver (IFAT) is a dynamically re-configurable array of integrate-and-fire neurons mimicking the physical properties and functionality of biological neurons. It's re-programmable capabilities enables many potential applications including use as a neural simulator and even visual processor for event-based ("spike-based"[85]) input/output. The IFAT PU implemented for these CMPs is further optimized for low-power and high neuron density.[86] This PU comprises of an array of 32x32 integrate-and-fire neurons

sharing a single synapse and soma per column (i.e. 32 shared synapses and 32 shared somas). Each synapse is designed using a switch-cap circuit in which the weight (or amount of charge integration) is proportional to the value of the 5-bit weight. It consists of a synaptic driving potential that allows the synapse to be either excitatory or inhibitory. The soma is designed using a shared comparator such that if the voltage of a neuron is greater than a preset threshold voltage, a spike is outputted. These spikes are represented by what are called address events (AEs). This event-based communication scheme allows for low-power weighted-sums and other neural network configurations enabling a wide-range of applications. Digital circuitry complementing these analog neurons is used for event generation, sending events to the neural array, and managing the output events.

6. **CID_PU** (designed by Gaspar Tognetti, Christos Sapsanis and Jonah Segupta) (1 PU slot, $\approx 0.7M$ transistors).

   The charge injection device (CID) array is 1 bit vector matrix multiplier which performs charge-based non-destructive computations using a DRAM memory.[87] The CID structure implemented on the CMPs comprises a 156x512 array of computational DRAM cells which carry out a 1-bit multiplication, i.e. logic AND based on a charge injection device. Each row in the array computes the inner product between an incident vector X and a weight vector Wi, obtaining the result in the charge domain. This charge is converted to a voltage

through a switched capacitor circuit and later converted to digital via a Sigma-Delta converter. That is, each row gives a 512x512 1-bit inner product in the charge domain, and is later converted to the digital, yielding a vector-matrix-multiplication. Noteworthy, both input vector X and weight matrix W may represent digital values in a unary representation, whether in a PWM fashion or a pseudo random pulse density.

7. **MLE_PU** (designed by Alejandro Pasciaroni, based on the original work by Andrew Cassidy) (15 PU slots, $\approx 33.5M$ transistors).

   This unit allows the computation of multivariate Gaussian Mixture models, and the Viterbi algorithm. This engine has three main blocks: the Gaussian Mixture Model block (GMM), the Cache memory and the Hidden Markov Model block (HMM). The GMM receives a vector of 39 dimensions and evaluates a set of distributions in the logarithmic domain. The resulting values are stored in the cache memory which acts as a buffer for outputting the values to either of the NoCs or to the HMM block. The HMM block computes the Viterbi algorithm for hidden Markov models. The most probable state of a hidden Markov model is computed based on a sequence of observed events. The block is able to compute 64 models in parallel, each of one having 3 states. Each state has an observation probability that is modeled by 16 Gaussian mixture distributions which are provided by the GMM block. Both blocks feature hierarchical memory, allowing memory transactions to be reduced, saving in latency and energy.

8. **NVM_PU** (designed by Gaspar Tognetti and Jonah Segupta) (1 PU slots, $\approx 1.9M$ transistors).

   The non-volatile-memory (NVM) processor is a mixed signal 128x512 computational memory arranged in a crossbar based on the esf3 SST cell. One-bit inputs x_i applied horizontally are multiplied by N-bit weights w_{ij}, which depend on the state of a memory cell (e.g. 4-bits). Later all multiplications are added along columns intrinsically on a current node, yielding an inner product. The output currents are converted to digital in parallel using 512 current-mode Sigma-Delta converters. Additionally, it is possible to feed any output of the array back to the input in a recurrent fashion.

9. **16VM_PU** (designed by Alejandro Pasciaroni) (1 PU slot, $\approx 3.2M$ transistors).

   The MAC unit performs vector-matrix multiplications. The matrix in this multiplication $M \in \mathbb{R}^{16x16}$, and its input vector $\vec{x} \in \mathbb{R}^{16}$. The unit is able to compute four vector-matrix multiplications simultaneously. Both matrix and vector can be loaded from either of the NoCs, and results can be forwarded to any of them as well. Additional to the result of the multiplication, an address can be forwarded with the data allowing to target certain local registers in the destination PU.

10. **10KC_PU** (designed by Kayode Sanni) (1 PU slot, $\approx 3.0M$ transistors).

    The 10KC_PU is an auxiliary unit that provides additional memory banks for

other processing units. Programmed through a small processor local to the PU through the *L2 network*, this cache PU operates as a standalone unit that can make read and write request to the main memory through the *L1 network*, or send data through *L2 network* to other processing unit. Each 10KC_PU comprises of 10KB of local memory with a read and write speed of 300MHz.

11. **VVM_PU** (designed by Kayode Sanni) (1 PU slot, $\approx 1.9M$ transistors).

    The VVM_PU computes inner products on an array of capacitors using mixed-signal stochastic circuits. By computing this product as charge in the analog domain, the energy cost of this computation can be scaled to thermal (kTC) noise limit, and hence minimize the energy cost to the order of femto-Joules. Moreover, a switch-capacitor ADC is used to decode this charge into a digital value with a trade-off of computation time and output precision. The full VVM_PU design comprises of 128 VVM cores that are capable of computing inner products with up to 9-element vectors at variable precision. Exploiting charge-based computing, each VVM can efficiently compute on the array at 1.8TMAC/W, and with the full VVM core at 28.6GMAC/W at roughly 8-bit precision. Additionally, this PU has a throughput of 225MOP/s, where an operation is an 8-bit MAC. Furthermore, this VVM_PU has been adapted for numerous of signal and image processing applications, including non-uniformity correction (NUC), fast-fourier transforms (FFT), neural networks, image filtering, and more.

12. **CMC_PU** (designed by Kayode Sanni) (1 PU slot, $\approx 2.7M$ transistors).

    The CMC_PU is a processor used for data manipulation by address remapping. Specifically, this unit can be used for image rotation and translation through destination-based remapping. Starting with destination addresses, this PU computes the corresponding source addresses from a rotation and translation matrix. After computing these new addresses, the processor reads and write data from the source destination based on the nearest-neighbor to the destination addresses. Optimized for 20,000 by 20,000 pixel images (400MPixels), each CMC_PU is capable of processing 120MP/s running at frequency of 300MHz.

13. **SP_PU** (designed by Alejandro Pasciaroni, based on the original work by Andrew Cassidy) (3 PU slots, $\approx 8.9M$ transistors).

    The SP_PU generates a 39 dimensions MFCC (Mel-frequency Cepstral Coefficients[88]) feature vector, having as input digital audio samples. Pre-filtering, FFT and DCT operations in the log-mel scale are some of the types of processing involved, which are implemented in pipeline fashion. This unit works with a timing window of 10 milliseconds named frame. In order to get more accurate features, the unit perform the processing on three consecutive overlapping frames with a sample rate of 16 KHz. The audio sample and each component of the resulting vector is represented by a 16-bit signed word. Dedicated inter-

faces has been designed to integrate the unit to the NoCs, so data can be sent or received to/from both networks depending on the unit configuration.

14. **PROG_PU** (designed by Tomás Figliolia, Kayode Sanni, Christos Sapsanis and Jamal Molin) (2 PU slots, $\approx 0.4M$ transistors).

    This unit performs the programming, as well as the debugging of the *DDR DRAM PHY*, locally generates the clocks used in the CMP, and provides locally generated biases used by mixed-signal PUs. Just like for both M0_PU and dual_M0_PU units, the signals used in the program and debug of the *DDR DRAM PHY* are additional to the ones corresponding to the four-phase handshaking protocol between the network node and the PU.

    This unit features four ring oscillators designed by Jamal Molin and Kayode Sanni, based on,[89] and two PLLs providing two programmable clocks each. A total of eight clock sources are the ones that can be used for both the high-speed memory interface and the NoCs. Upon a general reset is received on the network and the PUs, the default clock used across the whole CMP will be the one provided by the FPGA. A specialized asynchronous block called SEN_CLOCK was designed for the safe switching between two clock sources. A tree of these units was implemented for both network and *DDR DRAM PHY* sides of the chip, allowing any of the available eight clock sources to be used by either side of the chip. The configuration of the local clocks is done through

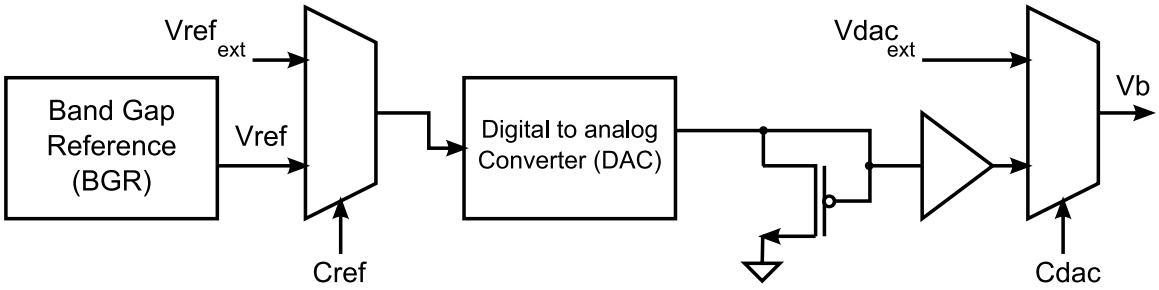


Figure 2.8: Block diagram for the local bias generators.

the *L2 network*.

A *Band Gap Reference* can be found in this PU, which is used for the current biases of the charge injection device (CID_PU) and vector-vector-multiplier (VVM_PU) processing units. The biasing consists of two parts: a *Bandgap Voltage Reference* (BGR) (based on[90]) and a *Programmable Current Biasing DAC* (based on[91]), followed by a voltage conversion driving two of the bias network grids. The BGR can provide a stable biasing point for a wide range of temperatures. The basic block diagram is presented in Figure 2.8. The external reference voltage $Vref_{ext}$ is provided as one of the external biases. The biases provided to both CID_PU and VVM_PU can be generated locally from two DACs, or external biases can be selected with the programming of the last analog multiplexer stage in Figure 2.8. A parasitic capacitance of $4nF$ and resistance of 310Ω in the biases' network grids have been extracted and used in the simulation of this unit.

The four assembled CMPs, using the different presented PUs, are shown in Figures 2.9, 2.10, 2.11 and 2.12. Due to the fact that many of the units work by counting,

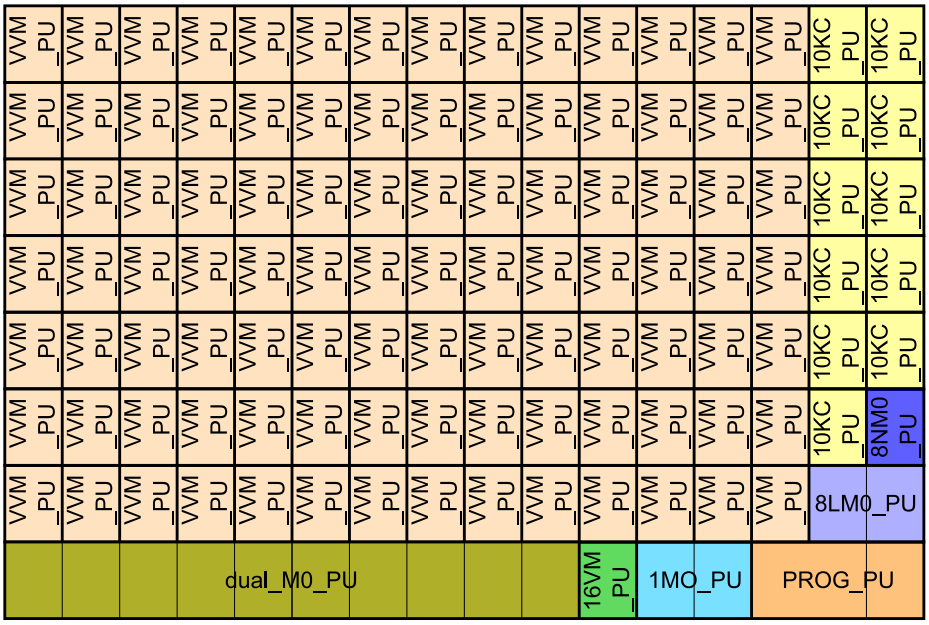


Figure 2.9: CMP1 *Yupana* PU breakdown.

the CMPs were named after the word *Abacus* in four different languages. CMP1 was named *Yupana*, from Inca's language, CMP2 was named *Salamis Tablet*, from the Greek language, CMP3 was named *Soroban*, from the Japanese language, and CMP4 was named *Suanpan*, from the Chinese language.

2.5 Power Distribution

The distribution of voltage supplies across all of the CMPs can be observed in Figures 2.13 and 2.14. Three are the main core voltage supplies, which are *VDD_PU*, *VDD_NET* and *VDD_DDR*. The first two are used in the NoCs' side of the chip where the network nodes and processing units are placed. In each processing unit one can see both *VDD_PU* and *VDD_NET* available for the PUs, and that is because the PU

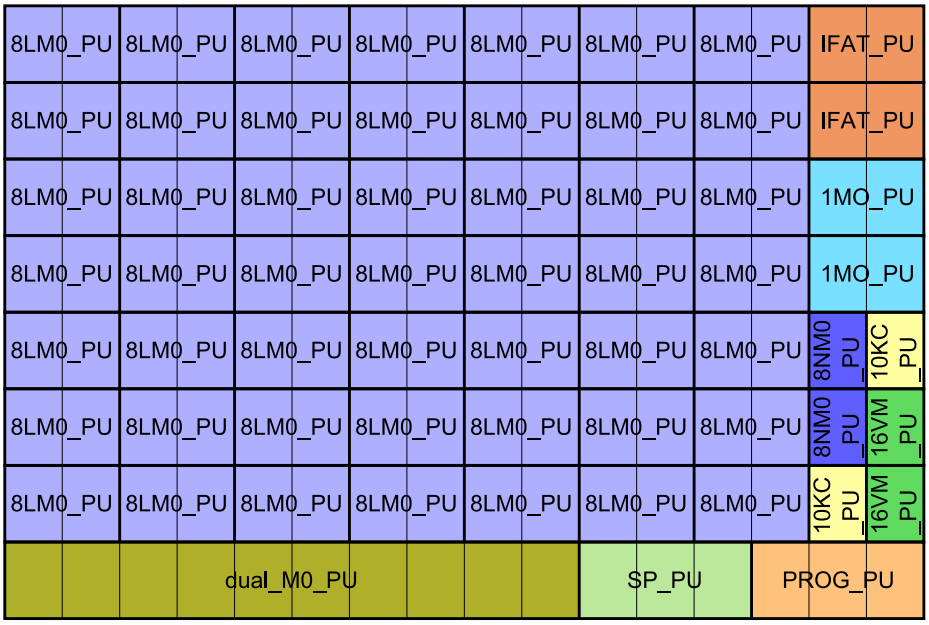


Figure 2.10: CMP2 *Salamis Tablet* PU breakdown.

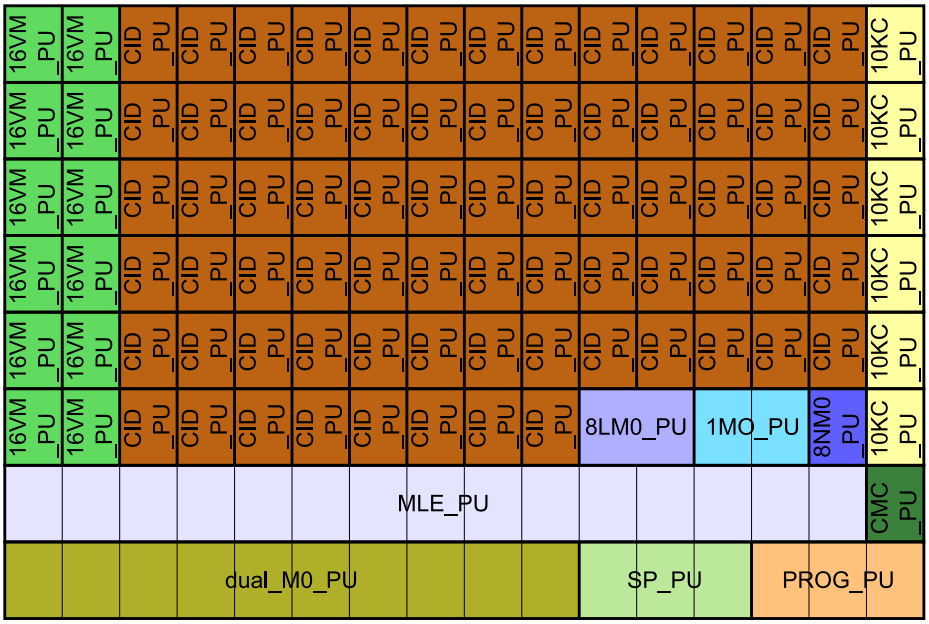


Figure 2.11: CMP3 *Soroban* PU breakdown.

designer can choose which of the voltage supplies is more convenient, according to the PU's clock speed requirements. On the high-speed memory interface side, both

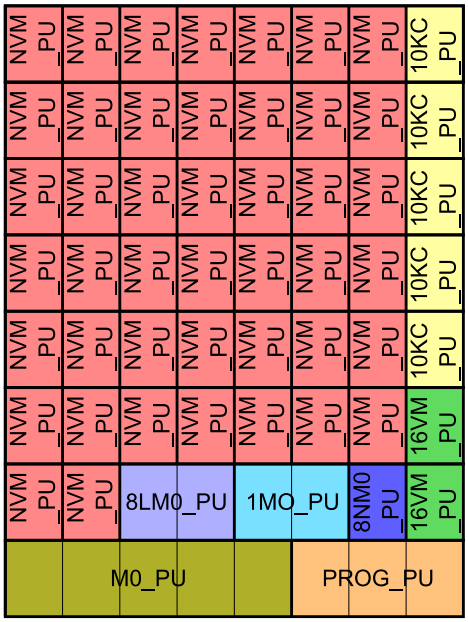


Figure 2.12: CMP4 *Suanpan* PU breakdown.

VDD_NET and *VDD_DDR* are being used. Voltage supply *VDD_NET* is mainly used on this side because of the necessity of level shifting the signals communicating the network with the *DDR DRAM PHY*.

With respect to the pad rings, they have been broken into two C-shape structures. The reason for this is not only because the core voltages for both the network side and *DDR DRAM PHY* side could be different, but also because the voltage levels used in the communication to the external memory could be different than the ones used in the communication with the FPGA as well. These two voltage supplies are the ones named as *VDD_E_DDR* and *VDD_E_FPGA*. All of the voltage supplies were designed to be set to $1.2V$, with the exception of *VDD_PU* which can be lower than $1.2V$. The division of all of the power domains allows any of them to be tuned individually without impacting any of the other domains.

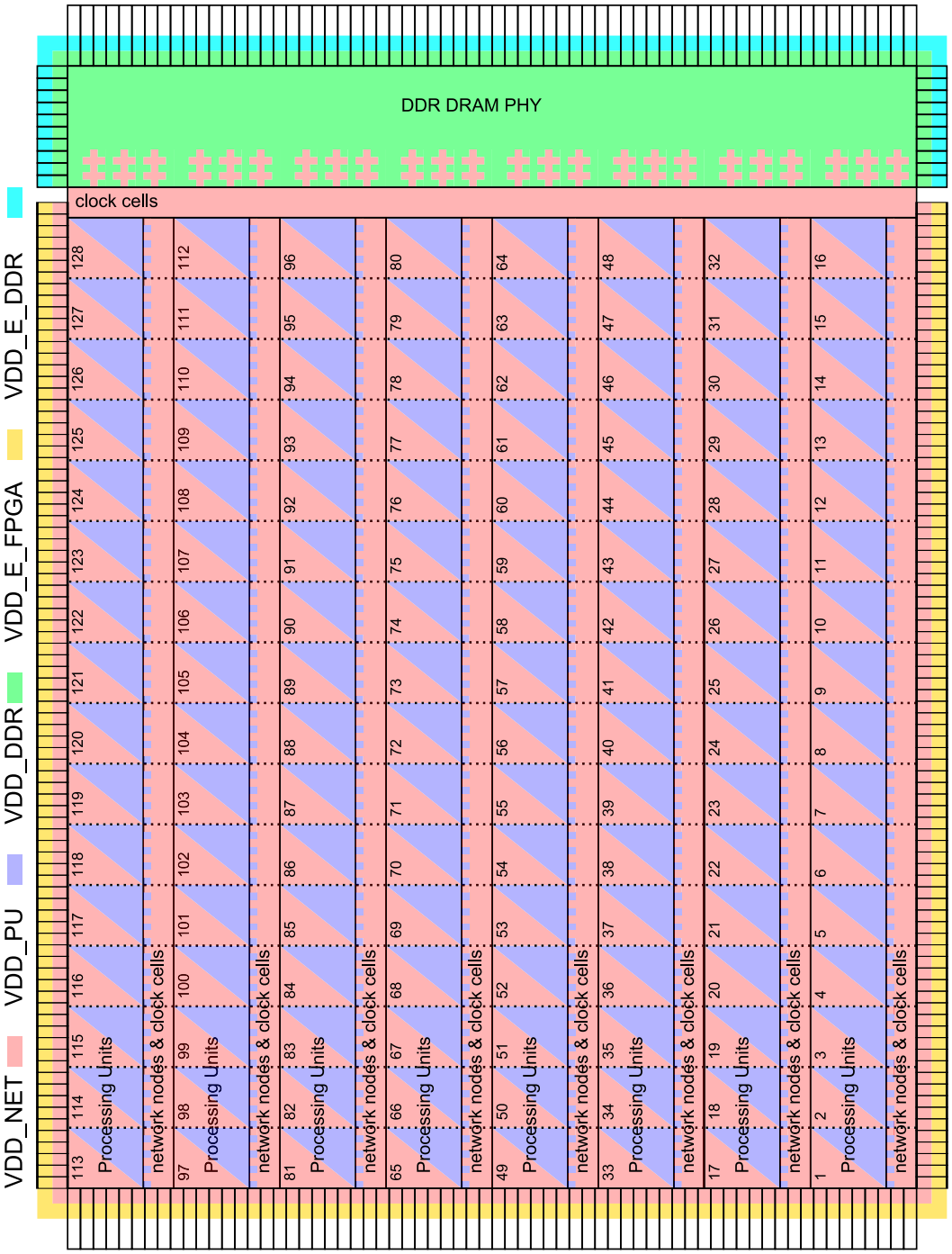


Figure 2.13: Voltage supplies across the 128 PUs CMP. The different voltage supplies are presented with different colors. One can see the pad rings are divided into two different c-sections because of the usage of different voltage supplies in the *DDR DRAM PHY* and the network. PUs can choose between using *VDD_NET* or *VDD_PU*.

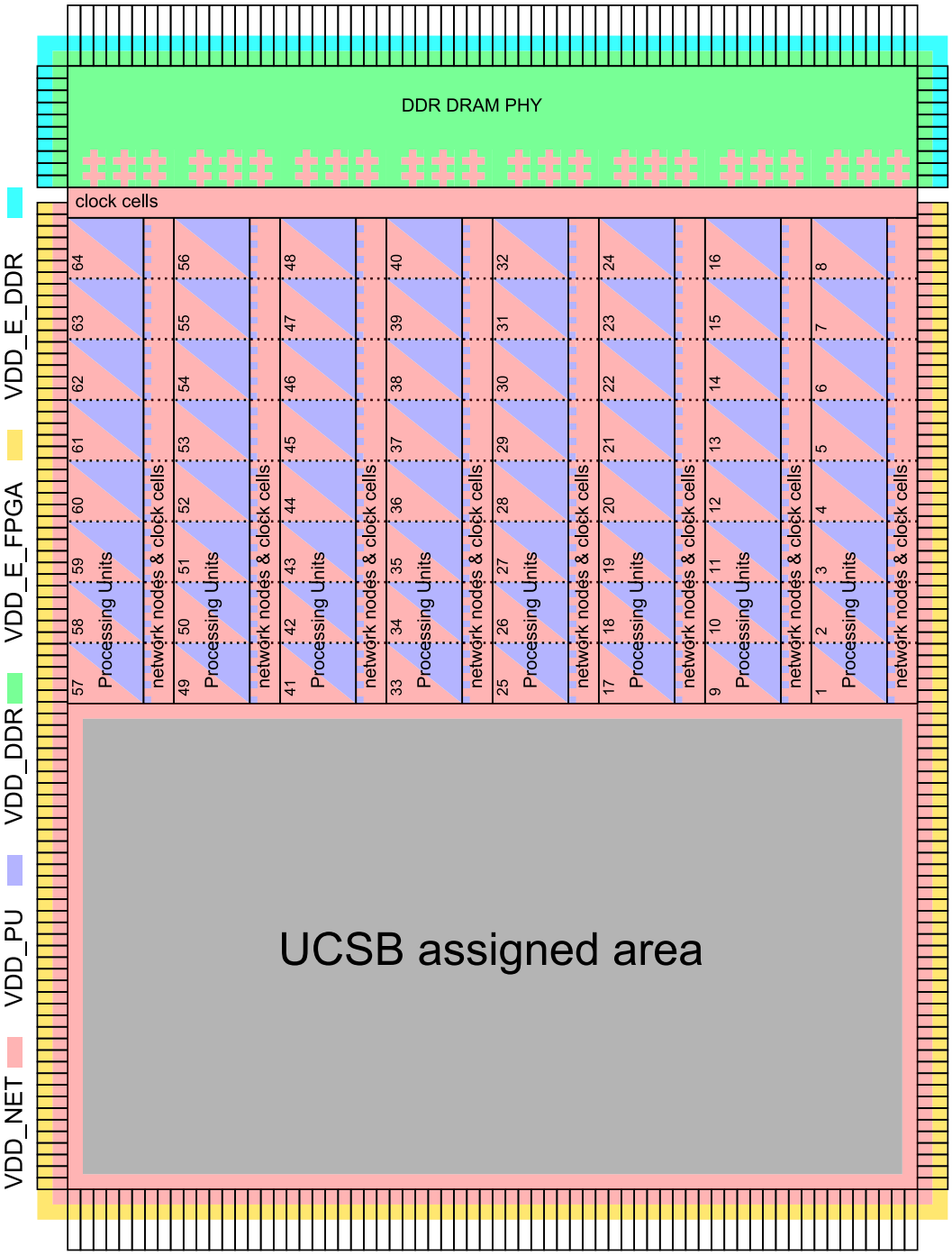


Figure 2.14: Voltage supplies across the 64 PUs CMP. The different voltage supplies are presented with different colors. One can see the pad rings are divided into two different c-sections because of the usage of different voltage supplies in the *DDR DRAM PHY* and the network. PUs can choose between using *VDD_NET* or *VDD_PU*.

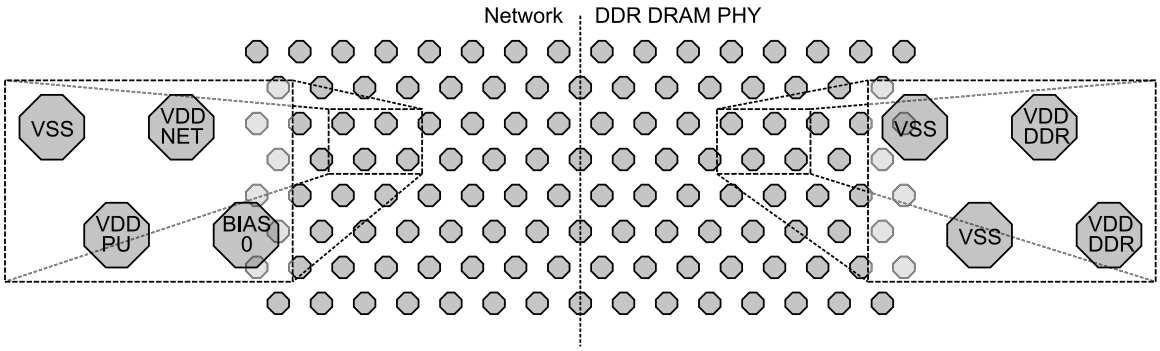


Figure 2.15: C4 bumps pattern for both Network and *DDR DRAM PHY* side. The pattern shown for both sides is repeated all over the CMP chips with the exception of the area used by UCSB.

All across the chips, C4 bumps were laid out supplying power to *VDD_NET*, *VDD_DDR* and *VDD_PU* domains. One can see the pattern used on both sides of the chip in Figure 2.15. In addition to the mentioned voltage supplies, one of the biases was also made available through these C4 bumps on the network side of the chip. The reason for doing this is that one of the PUs, whose processing is based on non-volatile memory (NVM), required an additional high voltage power supply (up to 2.5V). The gray block in Figure 2.14 is not covered by C4 bumps as it corresponds to the chip area assigned to UCSB.

As mentioned before, a combination of both C4 bumps and bondpads was available in every pad. By having this kind of redundancy, testing of these CMPs could be done without the need of the interposer chip. In Figure 2.16 the pads featuring both interfaces are shown.

Processing unit PROG_PU holds a *Band Gap Reference*. This unit is capable of generating local biases available to all of the PUs. A maximum of 16 are the possible

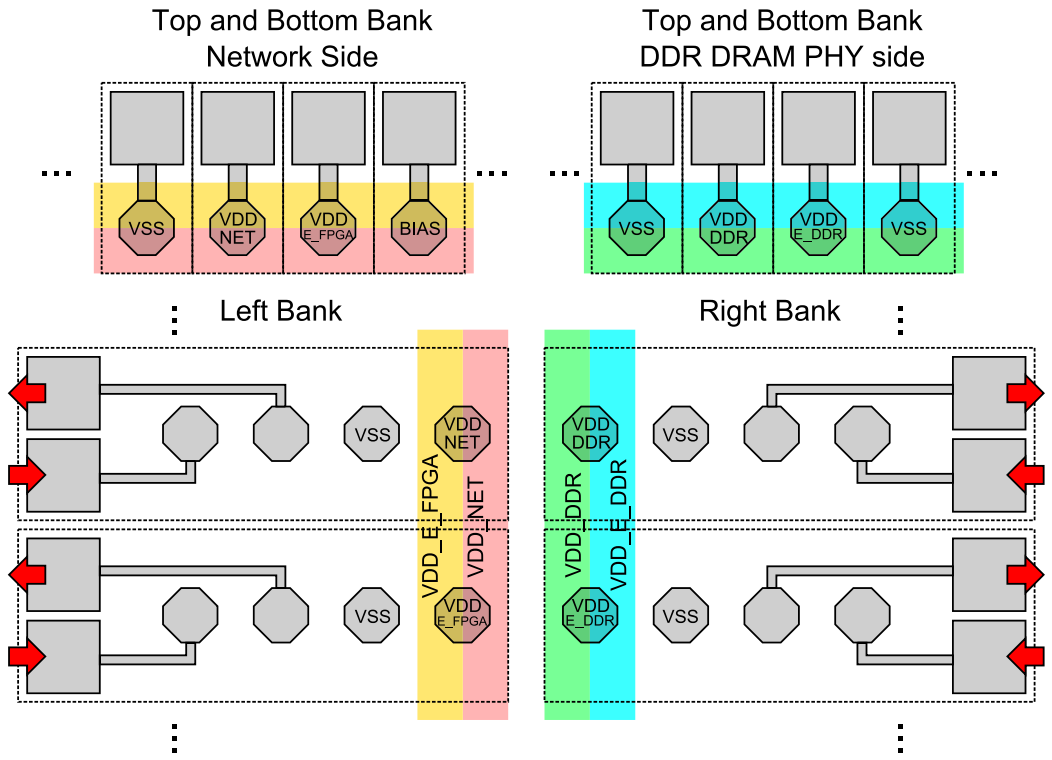


Figure 2.16: Combination of C4 bumps and bondpads. For the different pad banks in the CMP chips, each pad cell is composed of both C4 bumps and bondpads.

locally generated biases, and 16 are also the maximum number of biases provided externally. Table 2.1 shows the usage of the biases for some of the mixed signal processing units. In this case only two biases are generated locally, BIAS 14 and BIAS 15.

2.6 On-Chip Programmability of Clocks.

A low frequency clock will be received from the FPGA. Upon performing a general reset on chip after power-up, the PROG_PU unit will set this FPGA clock as the one used for both the *DDR DRAM PHY* and the network side of the CMP. Eight different

Bias	Type	CMP1	CMP2	CMP3	CMP4
BIAS 0	External	-	-	-	NVM_PU
BIAS 1 TO 13	Internal	-	-	-	-
BIAS 14	Internal	VVM_PU	-	CID_PU	-
BIAS 15	Internal	VVM_PU	-	CID_PU	-
BIAS 16	External	VVM_PU	IFAT_PU	CID_PU	NVM_PU
BIAS 17	External	VVM_PU	IFAT_PU	CID_PU	NVM_PU
BIAS 18	External	VVM_PU	IFAT_PU	CID_PU	NVM_PU
BIAS 19	External	VVM_PU	IFAT_PU	CID_PU	NVM_PU
BIAS 20	External	-	IFAT_PU	CID_PU	NVM_PU
BIAS 21	External	-	IFAT_PU	CID_PU	NVM_PU
BIAS 22	External	-	IFAT_PU	CID_PU	NVM_PU
BIAS 23	External	-	-	CID_PU	-
BIAS 24	External	-	-	CID_PU	-
BIAS 25	External	PROG_PU($Vext_a$)	PROG_PU($Vext_a$)	PROG_PU($Vext_a$)	PROG_PU($Vext_a$)
BIAS 26	External	PROG_PU($Vext_b$)	PROG_PU($Vext_b$)	PROG_PU($Vext_b$)	PROG_PU($Vext_b$)
BIAS 27	External	PROG_PU($Vref_i$)	PROG_PU($Vref_i$)	PROG_PU($Vref_i$)	PROG_PU($Vref_i$)
BIAS 28	External	PROG_PU($Vref_o$)	PROG_PU($Vref_o$)	PROG_PU($Vref_o$)	PROG_PU($Vref_o$)
BIAS 29	External	PROG_PU(PLL)	PROG_PU(PLL)	PROG_PU(PLL)	PROG_PU(PLL)
BIAS 30(HV)	External	-	-	-	NVM_PU(4.5V)
BIAS 31(HV)	External	-	-	-	NVM_PU(10.0V)

Table 2.1: Table of biases used by different PUs. Biases 16 to 31, and bias 0 are provided externally. Only two biases are generated locally, the 14 and 15.

clock sources are present in the PROG_PU unit. Four of them are generated by two local PLLs, and the other four are generated by custom designed ring oscillators. One cannot start using any of these clocks right away, as they require to be previously programmed. It is then required that upon power-up, after applying a global reset, the FPGA clock is the one selected, and remains being selected even after the global reset is deasserted. Both the ring oscillators and PLLs will then be configured through the *L2 network*, and once their output clock is assumed to be stable, a switch from the FPGA to any of the available clock sources is performed. This switch has to be done in a safe manner, and then a custom asynchronous block was designed in performing this clock switch.

When changing the source of a clock, one needs to first turn off the currently used clock, and turn on the new one at specific times where one knows will not create higher frequencies than the ones the circuits can handle. A block named *Pulse generator* responsible of generating pulses indicating when to turn off and on these clocks, is presented in Figure 2.17, with its corresponding timing diagram in Figure 2.18. Let's consider the currently used clock *clk1_i*, and the one to which it is desired to switch *clk2_i*. When the switch is desired to happen, input *sel_n_i* will be deasserted for a number of clock cycles. Considering independence between the two mentioned clocks and considering any possible clock source the one governing input *sel_n_i*, a series of falling-edge registers are added, allowing to carry out the clock domain crossing safely. After this, an edge detector will sense the change, and an RS flip-flop will set

to '0' its output *RS_Q*. Output signal *stop_o* will be responsible for this. This output is the one indicating when to turn off *clk1_i*. Registers are falling edge, and then when this output is asserted, one knows the clock source *clk1_i* has just been deasserted, making it a safe time to turn it off. It is worth mentioning that neither of the clock inputs *clk1_i* or *clk2_i* will be stopped at any time, they are directly connected the output of a clock source. It is in the next higher level of hierarchy that the clocks are manipulated before being distributed across the chip. The rising transition in output *stop_o* will mark the correct time to turn off *clk1_i*. The next important time is the one signaling the safe switch to clock *clk2_i*. All of the registers used in this block have a preset instead of reset, setting their output to '1' as reset input *reset_n_i* is deasserted. After *stop_o* has been asserted, and deasserted, a pulse will be generated for output *switch_o*, where the rising edge will once more identify the safe time to perform the clock change. Both *stop_o* and *switch_o* pulses cannot overlap, they have to be separated at least by 800ps. This condition has to be considered so that the clock sources are configured accordingly before performing the switch.

Figure 2.19 shows the block diagram of the two input clock switcher. Two *Pulse generator* blocks have been placed, where the clock inputs in one have been swapped in the other one. Two inputs will control the switch from one clock to the other. Input *sel1_i* will perform a change to the *clk1_i* clock, and *sel2_i* to the other one. The four outputs from the *Pulse generator* blocks will be fed to an asynchronous circuit, for which the state transition diagram is shown in Figure 2.20. The states in gray are the

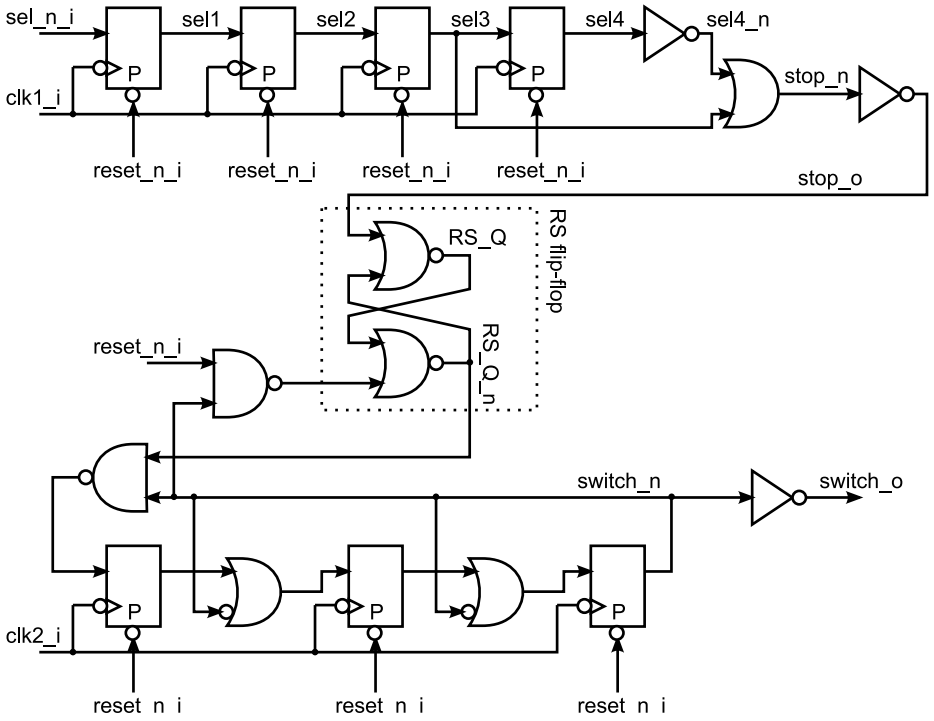


Figure 2.17: Architecture for *Pulse Generator* block. When switching from *clk1_i* to *clk2_i*, a rising edge in *stop_o* output will indicate the safe time to stop the first clock, and a rising edge in *switch_o* output the time when the switch should be done.

ones that will be decoded as the times where no clock should be selected. The states in red should allow *clk1_i* to be forwarded, and states in blue should allow the other clock. In this diagram one can observe that if a clock has already been selected, and mistakenly one selects to perform a change to that same clock, no clock switch will be performed. The *Decoder* output will use the state values in Figure 2.19 to turn off *clk_o* and to perform the switch safely. Due to the delay involved in performing the decoding, frequencies up to 2.0GHz were successfully simulated using MonteCarlo.

A successful way of performing a change from one clock to the other has been shown, but as mentioned before, eight are the locally generated clocks, plus the one

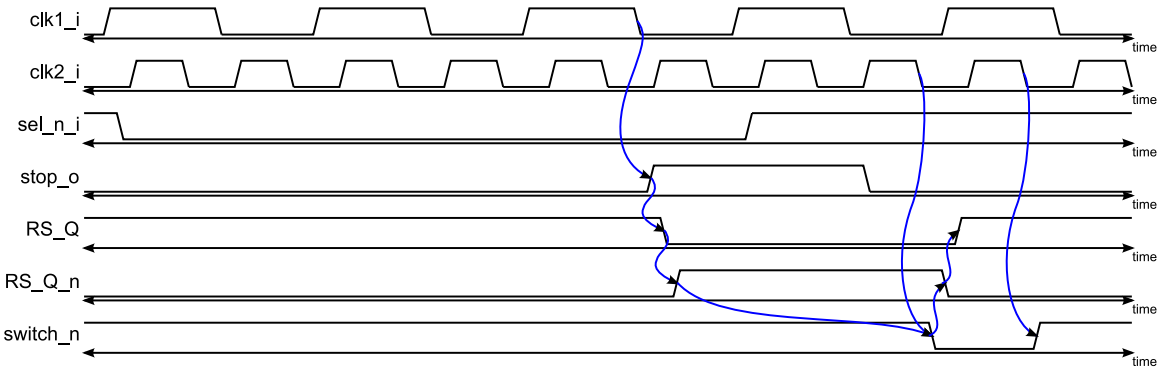


Figure 2.18: Timing diagram for internal signals of the *Pulse generator* block.

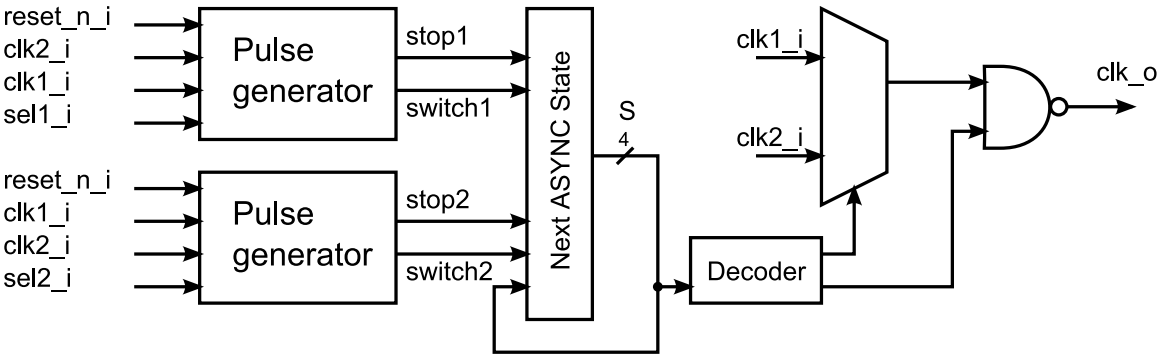


Figure 2.19: Block diagram for the two-input clock switcher cell.

coming from the FPGA. Figure 2.21 presents the tree of clock switchers used in the selection of one of the clock sources. Due to the fact that two are the clock sources needed for both network and memory interface side, two of these trees will be placed in the PROG_PU. After a global reset, the top most clock switcher will have the FPGA clock selected as the default one to forward to its output. By changing the values of the *selector* signals in sequence from the bottom to the top, any of the clock sources can be selected.

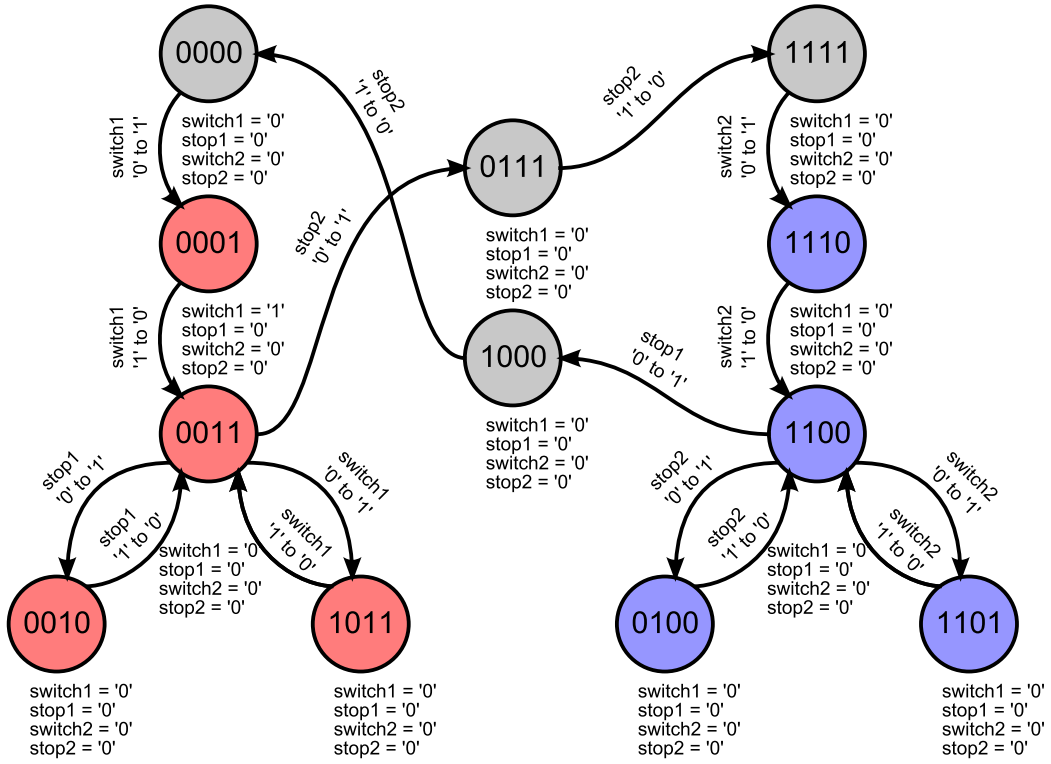


Figure 2.20: State diagram for the asynchronous circuit used in the clock switcher. States in gray signal the turn off of the output clock, states in red signal the usage of the *clk1_i* clock, and states in blue signal the usage of the other clock.

2.7 Stitching Logic

When putting together all of the CMPs, almost everything that had to be done was just the interconnection of all of the placed blocks such as the PUs, network nodes, clock tree cells (see Chapter 3) and the *DDR DRAM PHY*, with the exception of the synthesis of the *L2 network* node to which the FPGA connects to. No routing was necessary in between neighbor network nodes as it will be seen in Section 4.3 or in the connection of a PU to its network node, due to the fact that all of these blocks abut with each other. Signals that needed to be routed were:

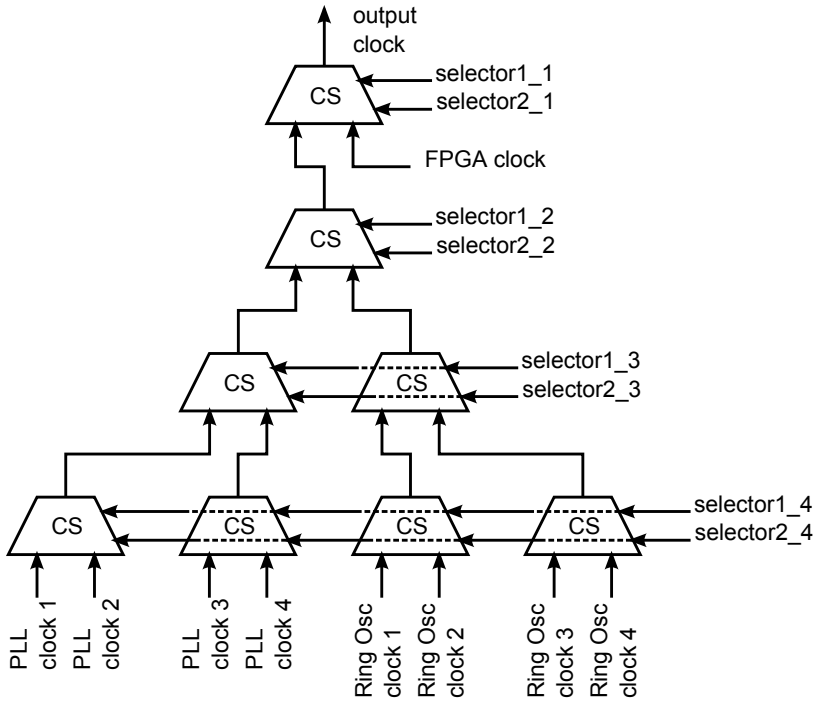


Figure 2.21: Clock switcher tree used in the selection of one in nine clocks sources in the PROG_PU unit. By sequentially changing the *selector* inputs from bottom to top, any of the nine clock sources can be used.

1. Connections in between the *DDR DRAM PHY* and the right-most column of network nodes on the chip.

2. Signals in the path going and coming back from the 3D DDR memory to the *DDR DRAM PHY* needed to be routed to their respective pads on the right side of the chip.

3. Signals used in the programming and debugging of the *DDR DRAM PHY* communicating the PROG_PU with the *DDR DRAM PHY* memory interface.

4. UART and Quad-SPI signals on the left bank of pads communicating to the M0_PU and dual_M0_PU.

5. The FPGA clock needed to be routed to the PROG_PU.

6. The general reset signal coming from the FPGA needed to be routed to one of the vertical clock cells distributing the signal throughout the whole network side of the chip.

7. Both network and *DDR DRAM PHY* clocks needed to be routed from the PROG_PU to the corresponding vertical clock cell in the network and the memory interface respectively.

8. The signal indicating the start of the self-diagnose for all of the network nodes needed to be routed to each single network node (see Chapter 4). The routing of this signal was not problematic as it is a very slow one.

9. All the pads providing both biases and power supplies needed to be routed to the low resistance network grids and the chip core rings, as well as the C4 bumps.

A *L2 network* node had to be synthesized for the connection of the FPGA to the *L2 network*. This network node did not feature a connection to the *L1 network* (allowing access to external memory), and only the EAST port was the one available in the connection to the *L2 network* (see Chapter 4). Compared to any other network node in the chip, this node was significantly smaller in area. The addition of an extra column of nodes just for the connection of the FPGA to the *L2 network* would have wasted a lot of area, and because this node is much smaller than any other node in

the chip, it was a good idea to perform its *logical synthesis* and *Place & Route* on the highest level of hierarchy for the chips. In order to avoid routing problems and to minimize routing distances, the FPGA network node was connected to node $(1, 0)$ for the case of the 64 PUs CMP, and $(1, 3)$ for the case of the 128 PUs CMP.

Because of the limited number of pads on the left side of the chip, the FPGA network node features a serializer and deserializer breaking up the *L2 network* packet into pieces of 39 bits. All of the pads on the left side of the CMPs were designed as bidirectional. Only 41 of those pads are effectively used as bidirectional, all of the other ones were hardwired to be either an input or output. Figure 2.22 shows the timing diagram for the exchange of two *L2 network* packets going into the *L2 network* and coming out of it. Signal *BUS_direction_i* is driven by the FPGA, and determines the configuration of the bidirectional pads. The assertion of signal *start_io* with the transmission of the first 39 bits indicates the beginning of an *L2 network* packet. The assertion of *send_io* will indicate the completion of the 281 bits (without considering the handshaking protocol signals) used in the transmission of data to and from a PU.

2.8 Final Layout Designs and Pinout

Figures 2.23, 2.24, 2.25 and 2.26 show the four CMP layouts with the respective transistor count. Table 2.2 presents the pinout of the chip, where the pad number increases clockwise.

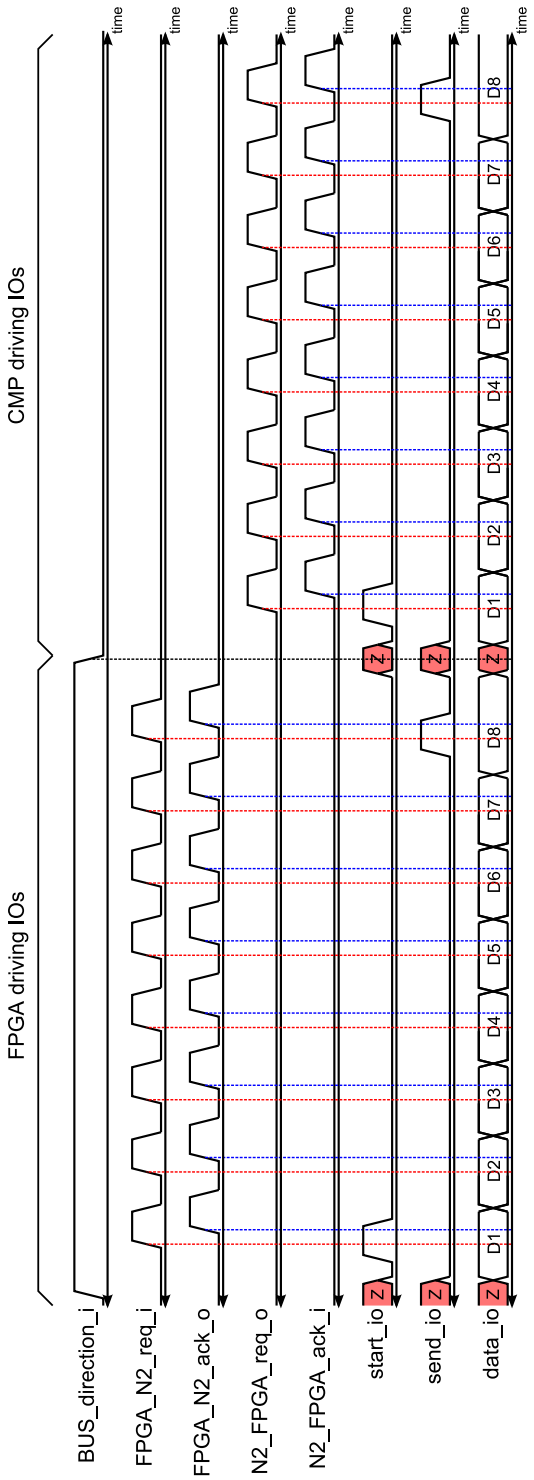


Figure 2.22: Timing diagram for the signals connecting the FPGA to the *L2 network*. With a '1' signal *BUS_direction_i* indicates that the FPGA is the one driving the bidirectional pads.

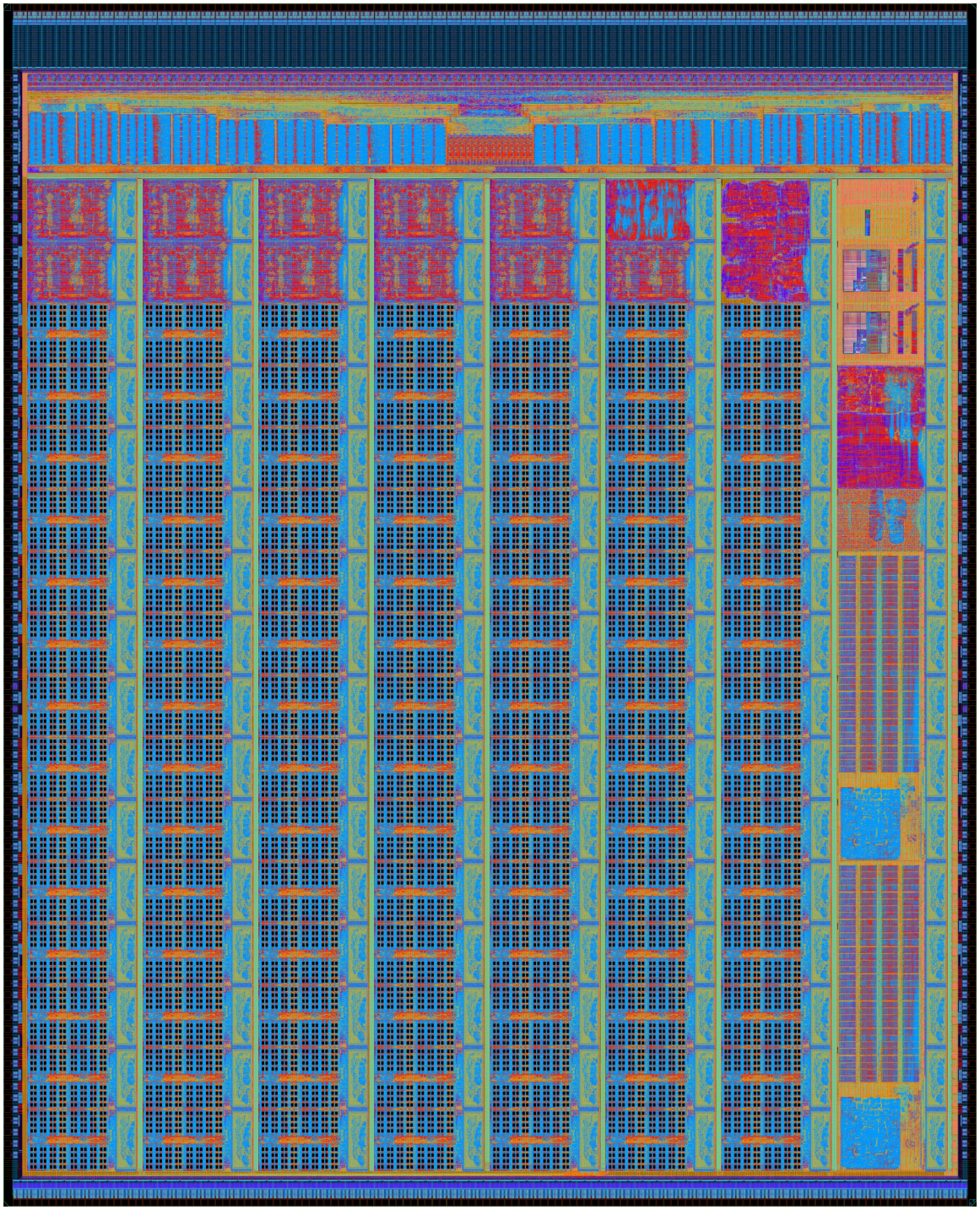


Figure 2.23: CMP1 *Yupana* layout ($\approx 385M$ transistors).

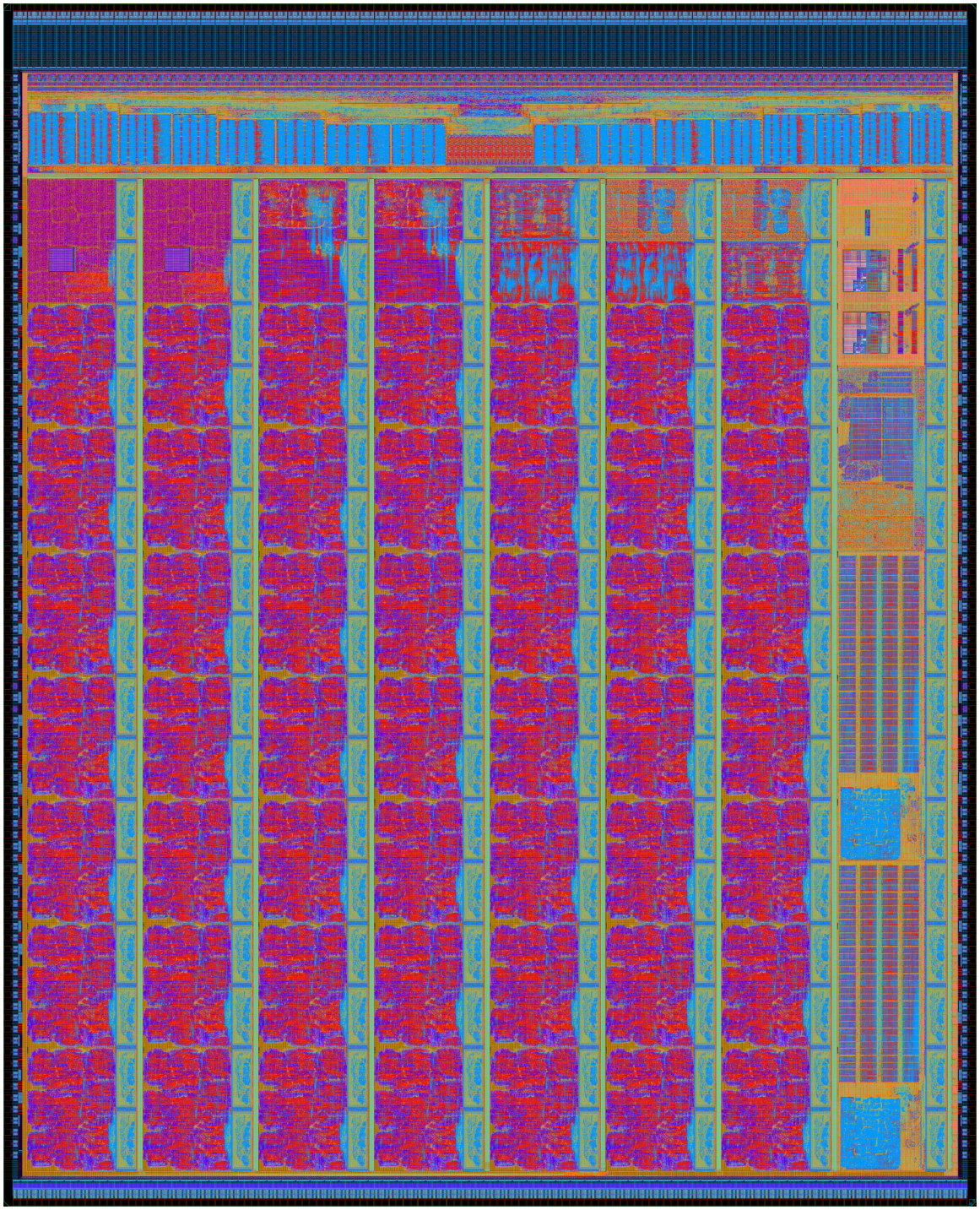


Figure 2.24: CMP2 *Salamis Tablet* layout ($\approx 454M$ transistors).

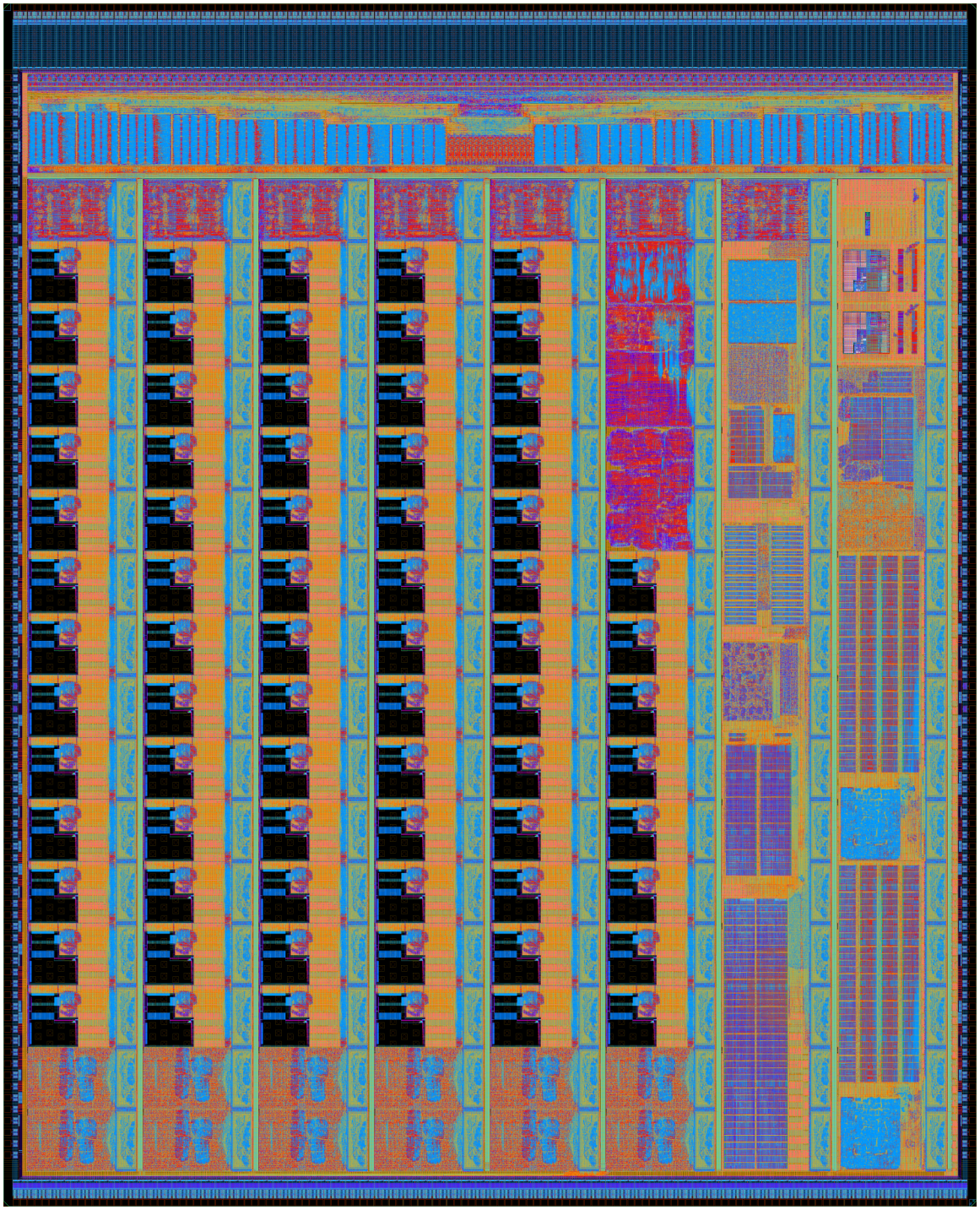


Figure 2.25: CMP3 *Soroban* layout ($\approx 320M$ transistors).

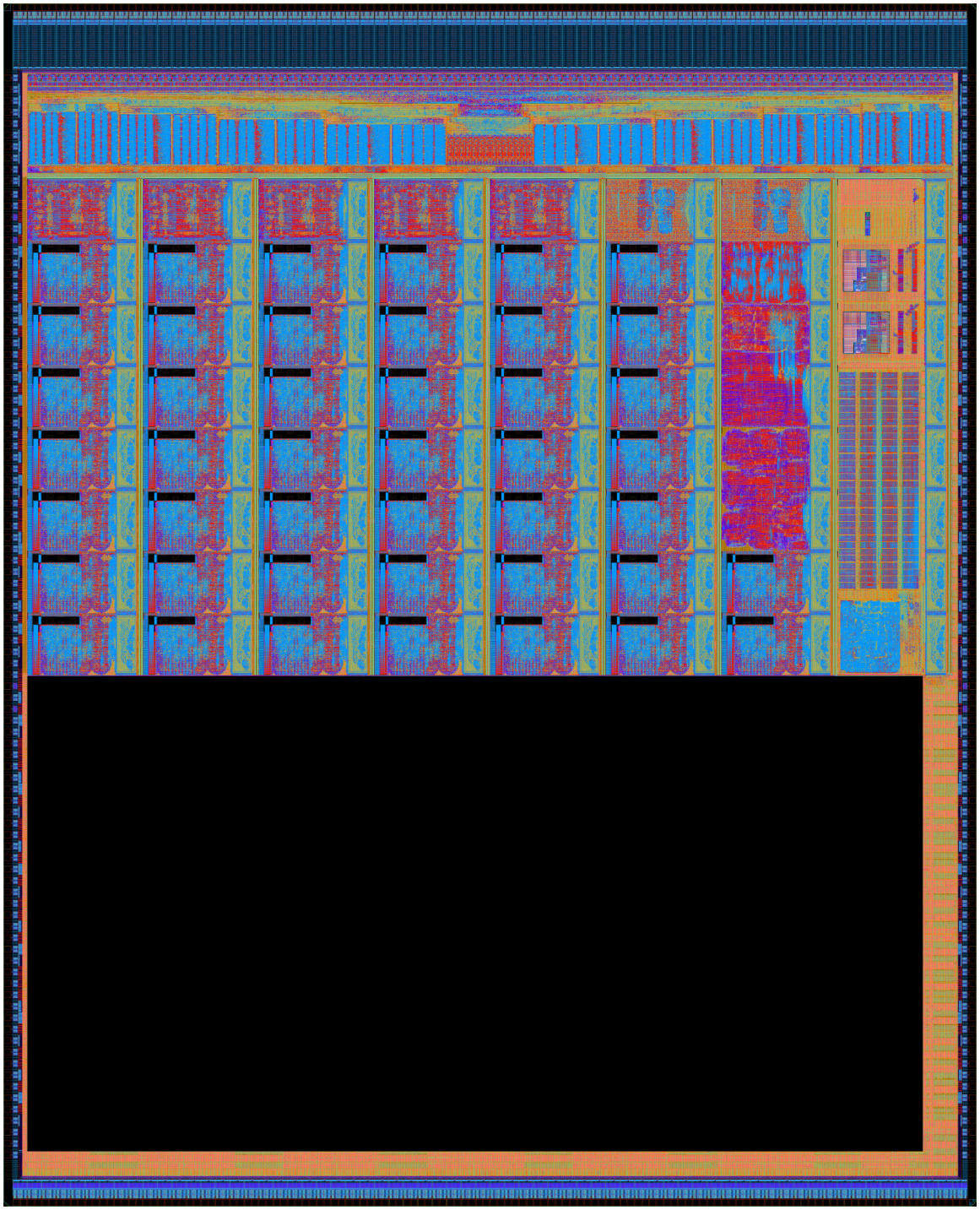


Figure 2.26: CMP4 *Suanpan* layout ($\approx 215M$ transistors).

Table 2.2: Bondpad signal assignment for all of the CMPs. The number assignment of the pads was done in a clockwise fashion.

Pad name	Pad #	Bank	Type	Description
FPGA_NoC_reset_i	0	Left	I	Asynchronous general reset for the network side of the chip.
QDSPI_tx_o(3 downto 0)	1 to 4	Left	O	Quad-SPI transmit wires from M0 processor number 0 to the FPGA.
QDSPI_rx_o(3 downto 0)	5 to 8	Left	I	Quad-SPI receive wires from the FPGA to M0 processor number 0.
QDSPI_cs_ld_o	9	Left	O	Quad-SPI chip select from the M0 processor number 0 to the FPGA.
QDSPI_clk_o	10	Left	O	Quad-SPI clock from the M0 processor number 0 to the FPGA.
UART_tx_1_o	11	Left	O	UART transmit wire from M0 processor number 1 to the FPGA.
UART_rx_1_i	12	Left	I	UART receive wire from the FPGA to the M0 processor number 1.
UART_rst_1_i	13	Left	I	UART reset wire from the FPGA to the M0 processor number 1.
UART_clk_1_i	14	Left	I	UART clock wire from the FPGA to the M0 processor number 1.
UART_tx_0_o	15	Left	O	UART transmit wire from M0 processor number 0 to the FPGA.
UART_rx_0_i	16	Left	I	UART receive wire from the FPGA to the M0 processor number 0.
UART_rst_0_i	17	Left	I	UART reset wire from the FPGA to the M0 processor number 0.
UART_clk_0_i	18	Left	I	UART clock wire from the FPGA to the M0 processor number 0.
N2_FPGA_ack_i	19	Left	I	Four-phase handshaking protocol acknowledge in the flow of packets out of the CMP.
N2_FPGA_req_o	20	Left	O	Four-phase handshaking protocol request in the flow of packets out of the CMP.
FPGA_N2_ack_o	21	Left	O	Four-phase handshaking protocol acknowledge in the flow of packets into the CMP.
FPGA_N2_req_i	22	Left	I	Four-phase handshaking protocol request in the flow of packets into the CMP.
BUS_direction_i	23	Left	I	Input signal configuring who can drive the inout signals *start_io*, *send_io* and *data_io*, the CMP or the FPGA.
start_io	24	Left	IO	If this signal is asserted during the transmission of a piece of a *L2 network* packet to or from the FPGA, this means that this is the first piece of the packet.
send_io	25	Left	IO	If this signal is asserted during the transmission of a piece of a *L2 network* packet to or from the FPGA, this means that this is the last piece of the packet.

data_io(38 downto 0)	26 to 64	Left	IO	Signal carrying data from a *L2 network* packet to or from the FPGA.
FPGA_link_down_o	65	Left	O	Every link in the *L2 network* is diagnosed, and the EAST link of the FPGA *L2 network* node is not an exception. This output indicates if there is any problem with that link.
FPGA_NoC_diagnose_i	0	Top	I	Signal indicating the start of the *L2 network* links' diagnose.
VDD_E_FPGA(power supply)	1, 2, 43, 84	Top	POWER	Power supply for the output pads driving connections to the FPGA.
VSS(ground)	3, 10, 17, 23, 30, 37, 44, 51, 58, 64, 71, 78, 85, 87, 89, 91, 93	Top	POWER	Ground pads.
BIAS_0(power supply)	8, 15, 22, 28, 35, 42, 49, 56, 63, 69, 76, 83	Top	POWER	External bias 0. This bias is actually a power supply used in CMP4.
BIAS_16(bias)	4, 45	Top	BIAS	External bias 16.
BIAS_17(bias)	6, 47	Top	BIAS	External bias 17.
BIAS_18(bias)	9, 50	Top	BIAS	External bias 18.
BIAS_19(bias)	11, 52	Top	BIAS	External bias 19.
BIAS_20(bias)	14, 55	Top	BIAS	External bias 20.
BIAS_21(bias)	16, 57	Top	BIAS	External bias 21.
BIAS_22(bias)	19, 60	Top	BIAS	External bias 22.
BIAS_23(bias)	21, 62	Top	BIAS	External bias 23.
BIAS_24(bias)	24, 65	Top	BIAS	External bias 24.
BIAS_25(bias)	26, 67	Top	BIAS	External bias 25.
BIAS_26(bias)	29, 70	Top	BIAS	External bias 26.
BIAS_27(bias)	31, 72	Top	BIAS	External bias 27.
BIAS_28(bias)	34, 75	Top	BIAS	External bias 28.
BIAS_29(bias)	36, 77	Top	BIAS	External bias 29.
BIAS_30(bias)	39, 80	Top	BIAS	External bias 30.

BIAS_31(bias)	41, 82	Top	BIAS	External bias 31.
VDD_NET(power supply)	5, 12, 18, 25, 32, 38, 46, 53, 59, 66, 73, 79	Top	POWER	Power supply for the network side of the CMP.
VDD_PU(power supply)	7, 13, 20, 27, 33, 40, 48, 54, 61, 68, 74, 81	Top	POWER	Power supply for the network side of the CMP.
VDD_E_DDR(power supply)	86, 90, 04	Top	POWER	Power supply for the output pads driving connections to the 3D DDR memory.
VDD_DDR(power supply)	88, 92	Top	POWER	Power supply for the *DDR DRAM PHY* side of the CMP.
DDR_CMP_clk_n_i & CMP_DDR_clk_n_o	0	Right	I/O	Negative clock in the path from external memory to the CMP, and vice-versa.
bits_DDR_i(0 to 63) & bits_HOST_o(0 to 63)	1 to 64	Right	I/O	64-bit buses in both ways from the CMP to external memory and vice-versa.
DDR_CMP_clk_p_i & CMP_DDR_clk_p_o	65	Right	I/O	Positive clock in the path from external memory to the CMP, and vice-versa.
VDD_E_DDR(power supply)	0, 4, 8	Bottom	POWER	Power supply for the output pads driving connections to the 3D DDR memory.
VSS(ground)	1, 3, 4, 7, 9, 16, 23, 30, 36, 43, 50, 57, 64, 71, 77, 84, 91	Bottom	POWER	Ground pads.
VDD_DDR(power supply)	2, 6	Bottom	POWER	Power supply for the *DDR DRAM PHY* side of the CMP.
VDD_E_FPGA(power supply)	10, 51, 92, 93	Bottom	POWER	Power supply for the output pads driving connections to the FPGA.
BIAS_0(power supply)	11, 18, 25, 31, 38, 45, 52, 59, 66, 72, 79, 86	Bottom	POWER	External bias 0. This bias is actually a power supply used in CMP4.
VDD_PU(power supply)	13, 20, 26, 33, 40, 46, 54, 61, 67, 74, 81, 87	Bottom	POWER	Power supply for the network side of the CMP.

VDD_NET(power supply)	15, 21, 28, 35, 41, 48, 56, 62, 69, 76, 82, 89	Bottom	POWER	Power supply for the network side of the CMP.
BIAS_16(bias)	49, 90	Bottom	BIAS	External bias 16.
BIAS_17(bias)	47, 88	Bottom	BIAS	External bias 17.
BIAS_18(bias)	44, 85	Bottom	BIAS	External bias 18.
BIAS_19(bias)	42, 83	Bottom	BIAS	External bias 19.
BIAS_20(bias)	39, 80	Bottom	BIAS	External bias 20.
BIAS_21(bias)	37, 78	Bottom	BIAS	External bias 21.
BIAS_22(bias)	34, 75	Bottom	BIAS	External bias 22.
BIAS_23(bias)	32, 73	Bottom	BIAS	External bias 23.
BIAS_24(bias)	29, 70	Bottom	BIAS	External bias 24.
BIAS_25(bias)	27, 68	Bottom	BIAS	External bias 25.
BIAS_26(bias)	24, 65	Bottom	BIAS	External bias 26.
BIAS_27(bias)	22, 63	Bottom	BIAS	External bias 27.
BIAS_28(bias)	19, 60	Bottom	BIAS	External bias 28.
BIAS_29(bias)	17, 58	Bottom	BIAS	External bias 29.
BIAS_30(bias)	14, 55	Bottom	BIAS	External bias 30.
BIAS_31(bias)	12, 53	Bottom	BIAS	External bias 31.
FPGA_NoC_clk_i	94	Bottom	I	Clock supplied by the FPGA.

2.9 Power Up Sequence and Configuration

During the power-up sequence, configuration packets will have to be sent to the PROG_PU, and then Table 2.3 presents the different configuration packets required to configure and debug the *DDR DRAM PHY*, to program the local *Band Gap Reference*, and to configure the ring oscillators and PLLs. The power-up sequence is as following:

1. Right after powering the CMP, a reset pulse is sent by the FPGA to the CMP through the input pad *FPGA_NoC_reset_i*. With this reset, both of the network and memory interface parts of the CMP will take their clock signal from the FPGA supplied clock through input *FPGA_NoC_clk_i*, selected by PROG_PU. This clock is expected not to be higher than 100MHz. The reset pulse is recommended to have a duration of a few clock cycles (at least 3) from the clock supplied by the FPGA. This reset pulse will set both NoCs to a known state, and will also be forwarded to the PUs in case a mixed-signal PU, like PU_NVM, requires it.

2. A pulse needs to be sent through input *FPGA_NoC_diagnose_i* (this is a very slow signal and requires a pulse duration of at least 8 FPGA clock cycles). This will command all of the network nodes to diagnose the links to their neighbors. Because of the large number of signals connecting two network nodes, one had

to come up with a scheme to diagnose the correct functioning of every single network interface. This will allow the local routing tables in a node to be modified automatically if any interface happens to not work properly.

3. One needs to now identify which of the PUs are reachable, due to the fact that some network links could be down, inhibiting certain PUs to be usable. Consequently, the FPGA needs to send a ping packet to every single PU. If the target PU is reachable, then a ping answer will be received by the FPGA. This ping answer will additionally contain information about the state of the local connections to the neighboring nodes. If a PU is unreachable, the sent ping packet will keep circulating in the *L2 network* until its *time counter* overflows, in which case the packet is dropped. The *time counter* is addressed in Chapter 4, and it represents the time a packet has been circulating in the *L2 network*, increasing by one count for each hop the packet suffers.

4. If PROG_PU was successfully reached, packets are sent from the FPGA to the PROG_PU configuring the local PLLs and ring oscillators. After this, assuming stability at the output of the local clock sources, *packet 0* from Table 2.3 is sent to the PROG_PU, configuring the clock switcher tree. This will allow to select any clock source for either the network or the *DDR DRAM PHY*. After the safe clock switch was perform, one can additionally send another packet to configure the local PLLs and ring oscillators, to speed up their frequency if desired.

5. The configuration for the local *Band Gap Reference* can be set with *packet 18*.

6. The FPGA will now have to send a packet to each reachable PU with the local clock configuration. This local clock is generated from the selected clock source in the PROG_PU.

7. After the PU local clocks are configured, and considered to have settled, then a reset packet is sent to every reachable PU. With this packet one can select the reset pulse duration.

8. With the general reset through input signal *FPGA_NoC_reset_i*, the communication between a PU and its network node is disabled. This is done so that if a PU is malfunctioning, then this will prevent it from filling the NoCs with unwanted traffic. If one knows which of the PUs are working correctly, then a packet to each of these PUs is sent from the FPGA, enabling the PUs to send and receive packets.

9. At this point the network side of the CMP is functional, and then the *DDR DRAM PHY* needs to be configured. The *DDR DRAM PHY* clock has already been configured, but now both local clock trees seen in Figure 3.5 need to be reset. This is done by sending *packet 1* and *packet 2* to PROG_PU.

10. After this, all of the internal blocks to the *DDR DRAM PHY* need to be reset in sequence. The FPGA needs to send *packet 3*, *packet 4* and *packet 5* in sequence.

11. As it will be mentioned in Chapter 5, in the path coming from external memory, programmable delay lines will be trained to ensure the correct reception of bits from the 3D DDR. One can configure manually the programmable delays in the interface to the 3D DDR memory, or one can opt to have them trained. At this point, if desired, the programmable delay values can be set by sending *packet 6* to the PROG_PU. If one will use these programmed delays, then *packet 7* is sent to PROG_PU indicating the usage of these values. After this, *packet 8* is sent to PROG_PU enabling the transmission of the training sequence (configured with *packet 22*). After the external memory has finished configuring its programmable delays, *packet 9* is transmitted indicating the stop in the transmission of the training sequence. During the transmission of the training sequence, one can observe the values being latched in every single line coming from external memory. One can command to withdraw a sample from a particular line with *packet 12*. For reading this sample, *packet 17* needs to be sent to PROG_PU.

12. If training is desired to be locally performed in the *DDR DRAM PHY*, the previous step can be skipped. One can send *packet 10* indicating the usage of the trained values for the programmable delays. After this, transmission of *packet 11* will start the training process. This will automatically transmit the training sequence to external memory, perform the required training, and will stop the transmission of the training sequence once the equalization has finished.

By sending *packet 17* one can sense the signals indicating if a successful training was performed.

13. After the training process has finished, *packet 13* and *packet 14* need to be sent. The equalization of the input pads will probably have triggered frame and parity errors (see Chapter 5), that are cleared by sending the mentioned packets. Parity and frame sequence errors can be read from PROG_PU with *packet 17*. The frame sequence can be configured with *packet 22*.

14. Because the *L1 network* is already configured, and the training has already finished, *packet 15* and *packet 16* need to be sent to PROG_PU. This will enable to forward packets from external memory to the *L1 network* and will allow the communication of every *L1 network* token-ring with the local buffers in the *DDR DRAM PHY*. With *packet 19* one can customize even more the interface between the *DDR DRAM PHY* and the *L1 network*.

Table 2.3: Configuring and debugging packets sent to PROG_PU. Field *data* corresponds to the 256 bits of data sent in the *L2 network* packet to the PROG_PU. *is_data* identifies if the packet is a control or data packet. *reg_addr* corresponds to one of the fields in the *L2 network* packet addressing local registers in the target PU. For a better understanding of this table, please see Chapter 4 and Chapter 5.

Packet #	is_data (1 bit)	data (256 bits)	reg_addr (10 bits)	Description
0	'0'	'x0...01'	'xXXX'	*Clock switcher tree configuration. data(8 downto 5)* and *data(4 downto 1)* configure the *selector* signals in Figure 2.21 for both the network side and the *DDR DRAM PHY* side of the CMPs respectively. *data(16 downto 9)* configures the pulse width for those *selector* signals and the waiting time in moving on to the configuration of the next level in the clock switcher tree.
1	'0'	'x0...02'	'xXXX'	This packet pulses input *reset_HOST_clk_i* from the *DDR DRAM PHY*. This resets the clock tree cell in the flow to external memory.
2	'0'	'x0...04'	'xXXX'	This packet pulses input *reset_DDR_clk_i* from the *DDR DRAM PHY*. This resets the clock tree cell in the flow coming from external memory.
3	'0'	'x0...08'	'xXXX'	This packet pulses input *reset_PI_i* from the *DDR DRAM PHY*. This resets the *Port_Interface* blocks (Section 5.6).
4	'0'	'x0...010'	'xXXX'	This packet pulses input *reset_MD_i* from the *DDR DRAM PHY*. This resets the *Mux_Demux* block (Section 5.5).
5	'0'	'x0...020'	'xXXX'	This packet pulses input *reset_PA_i* from the *DDR DRAM PHY*. This resets the *PADS_alignment* block (Section 5.4).
6	'0'	'x0...040'	'xXXX'	This packet pulses the write enable input signal *param_we_i* from the *DDR DRAM PHY*. *data(37 downto 32)* will address one of the 64 *PAD_interface* blocks (Section 5.3). On the target block, *data(43 downto 38)* will set input *p_clock_delay_HOST_i*, *data(49 downto 44)* will set input *p_data_delay_HOST_i*, and *data(52 downto 50)* will set input *p_align_HOST_i*.
7	'0'	'x0...080'	'xXXX'	This packet pulses the input signal *programmed_HOST_i* from the *DDR DRAM PHY*. This indicates that the *PAD_interface* blocks (Section 5.3) should use the programmed delays instead of the trained ones.

8	'0'	'x0...0100'	'xXXX'	This packet pulses the input signal *send_seq_ov_HOST_i* from the *DDR DRAM PHY*. This indicates that the *PAD_interface* blocks (Section 5.3) should transmit the training sequence to the external memory and no local training will be done. The delays will be programmed by the user.
9	'0'	'x0...0200'	'xXXX'	This packet pulses the input signal *stop_seq_ov_HOST_i* from the *DDR DRAM PHY*. This indicates the *PAD_interface* blocks (Section 5.3) to stop the transmission of the training sequence to the external memory. This is used only if signal input *send_seq_ov_HOST_i* from the *DDR DRAM PHY* was previously pulsed.
10	'0'	'x0...0400'	'xXXX'	This packet pulses the input signal *trained_HOST_i* from the *DDR DRAM PHY*. This indicates that the *PAD_interface* blocks (Section 5.3) should use the trained delays instead of the programmed ones.
11	'0'	'x0...0800'	'xXXX'	This packet pulses the input signal *send_seq_HOST_i* from the *DDR DRAM PHY*. This indicates that the *PAD_interface* blocks (Section 5.3) should start the training process for the external memory input pads.
12	'0'	'x0...01000'	'xXXX'	This packet pulses the input signal *sample_en_HOST_i* from the *DDR DRAM PHY*. This signal will allow a 16 consecutive bits sample to be taken from one of the 64 bitlines coming from external memory, addressing one of the *PAD_interface* blocks (Section 5.3) with *data(37 downto 32)*. This is done for debugging purposes for the training of the input pads.
13	'0'	'x0...02000'	'xXXX'	This packet pulses the input signal *reset_p_error_HOST_i* from the *DDR DRAM PHY*. This will clear the one bit and two bit error indicators *one_error_HOST_o* and *two_error_HOST_o* coming out of the *PADS_alignment* block (Section 5.4). These two outputs analyze the existence of parity bit errors in the packets received from the external memory.
14	'0'	'x0...04000'	'xXXX'	This packet pulses the input signal *reset_f_error_HOST_i* from the *DDR DRAM PHY*. This will clear output signal *frame_error_HOST_o* coming out of the *PADS_alignment* block (Section 5.4), indicating the lost of the frame sequence from external memory.
15	'0'	'x0...08000'	'xXXX'	This packet pulses the input signal *allow_data_HOST_i* from the *DDR DRAM PHY*. When the *PAD_interface* blocks are reset, all zeros are sent from the *PAD_interface* blocks to the *PADS_alignment* block. It is only with a pulse at input *allow_data_HOST_i*, that the data from external memory coming out of *PAD_interface* blocks (Section 5.3) is forwarded.

16	'0'	'x0...010000'	'xXXX'	This packet pulses the input signal *allow_data_PI_i* from the *DDR DRAM PHY*. Upon reset, blocks *Port_Interface* (Section 5.6) do not allow any packets to be received or sent from and to the *L1 network*. By pulsing *allow_data_PI_i*, the communication of the *L1 network* and the *DDR DRAM PHY* is enabled.
17	'1'	'xX...X'	'x0...00'	This packet triggers a packet to be sent to the destination set by *data(8 downto 6)* for the vertical address and *data(5 downto 1)* for the horizontal address in the *L2 network*. For the case *data(10 downto 9) = '00'*, signal *dropped_cnt_DDR_o(10 bits)* is transmitted back. For the case *data(10 downto 9) = '01'*, signals *frame_error_HOST_o*, *two_error_HOST_o*, *one_error_HOST_o*, *sample_HOST_o(16 bits)* and *t_align_HOST_o(3 bits)* are transmitted back. For the case *data(10 downto 9) = '10'*, signals *t_data_delay_HOST_o(6 bits)*, *t_clock_delay_HOST_o(6 bits)*, *train_complete_HOST_o*, *train_succeded_HOST_o*, *align_complete_HOST_o*, *align_succeded_HOST_o* and *N1_empty_o(8 bits)* are transmitted back. All of the output signals transmitted belong to the *DDR DRAM PHY*, with the exception of *N1_empty_o* which senses if any of the *L1 network* token rings are empty.
18	'1'	'xX...X'	'x0...01'	Configuration for the local Band Gap reference.
19	'1'	'xX...X'	'x0...02'	*data(5 downto 0)* configures the maximum number of words written to the buffer allocating transactions from the *L1 network* to be sent to external memory in the *Port_interface* blocks (Section 5.6). *data(15 downto 6)* configures the number of clock cycles of inactivity on those buffers that will trigger the transmission to external memory.
20	'1'	'xX...X'	'x0...03'	*data(15 downto 0)* will configure the number of clock cycles for Packet 1. *data(31 downto 16)* will configure the number of clock cycles for Packet 2. *data(47 downto 32)* will configure the number of clock cycles for Packet 3. *data(63 downto 48)* will configure the number of clock cycles for Packet 4. *data(79 downto 64)* will configure the number of clock cycles for Packet 5.

21	'1'	'xX...X'	'x0...04'	*data(15 downto 0)* will configure the number of clock cycles for Packet 6. *data(31 downto 16)* will configure the number of clock cycles for Packet 7. *data(47 downto 32)* will configure the number of clock cycles for Packet 8. *data(63 downto 47)* will configure the number of clock cycles for Packet 9. *data(79 downto 64)* will configure the number of clock cycles for Packet 10. *data(95 downto 80)* will configure the number of clock cycles for Packet 11. *data(111 downto 96)* will configure the number of clock cycles for Packet 12. *data(127 downto 112)* will configure the number of clock cycles for Packet 13. *data(143 downto 128)* will configure the number of clock cycles for Packet 14. *data(159 downto 144)* will configure the number of clock cycles for Packet 15. *data(175 downto 160)* will configure the number of clock cycles for Packet 16.
22	'1'	'xX...X'	'x0...05'	*data(15 downto 0)* will configure *train_seq_HOST_i* (16 bit training sequence). *data(31 downto 16)* will configure *frame_seq_HOST_i* (expected frame sequence from external memory). *data(37 downto 32)* will target one of the 64 *PAD_interface* blocks used in the programming of the delays.

Chapter 3

Clock Tree Design

3.1 Clock Tree Usual Solutions

The design of a reliable clock tree, especially for high frequencies, is not a trivial matter. When large silicon areas need to be fed with the same clock, several clock tree levels need to be used, and a choice usually needs to take place regarding what is found to be more important, skew or slew.[17] High slew is desirable because the relative error $Slew_{error}/Period$ is lower, and this will have a lower impact in the maximum operating frequency of the clock. Let's assume a $1GHz$ clock tree is to be built, a reasonable slew for all the nets in the clock tree could be $80ps$. If for some reason one of the clock tree drivers is affected by mismatch and it reaches slower slew by 50%, this is not very problematic, instead of $1ns$ period, the period will be $1.04ns$. The clock speed is reduced by only $\approx 40MHz$. If on the other hand a $200ps$

slew is chosen, then, using the same mismatch, not $40MHz$ but $100MHz$ will be the speed reduction the clock will suffer. Some of the possible solutions to this problem are the implementation of H or Fishbone clock trees[18] (see Figure 3.1). These clock trees can be synthesized with *Place & Route* tools, and they usually achieve good skew and slew. These structures rely on very powerful clock tree drivers and on specific placement for these drivers and the wires connecting them. Geometry is very important to these designs, and the problem they carry is that usually flat *Place & Route* needs to take place, because the positions of the clock drivers and wires are not easily changeable.

In the design of an H tree, from Figure 3.1, it can be seen that as the number of levels in the clock tree is increased, the areas that can possibly be used as blocks in a hierarchical design become smaller and smaller (area in green), making a flat synthesis the only viable option for most cases. On the other hand, the Fishbone clock tree gives much more freedom if modularity is desired in a design. Some of the problems this design has, are that *Place & Route* tools are usually not very efficient and a lot of silicon area is wasted, making it more convenient to design these clock trees in a custom manner using cad tools. The Fishbone tree seems to be a good option for the clock distribution network required for the CMPs, but unfortunately it does not deal properly with skew at the output drivers. Every column of drivers has its inputs shorted all together as well as its outputs, but not every driver output sees the same impedance in the line, making it very difficult to achieve a very low skew.

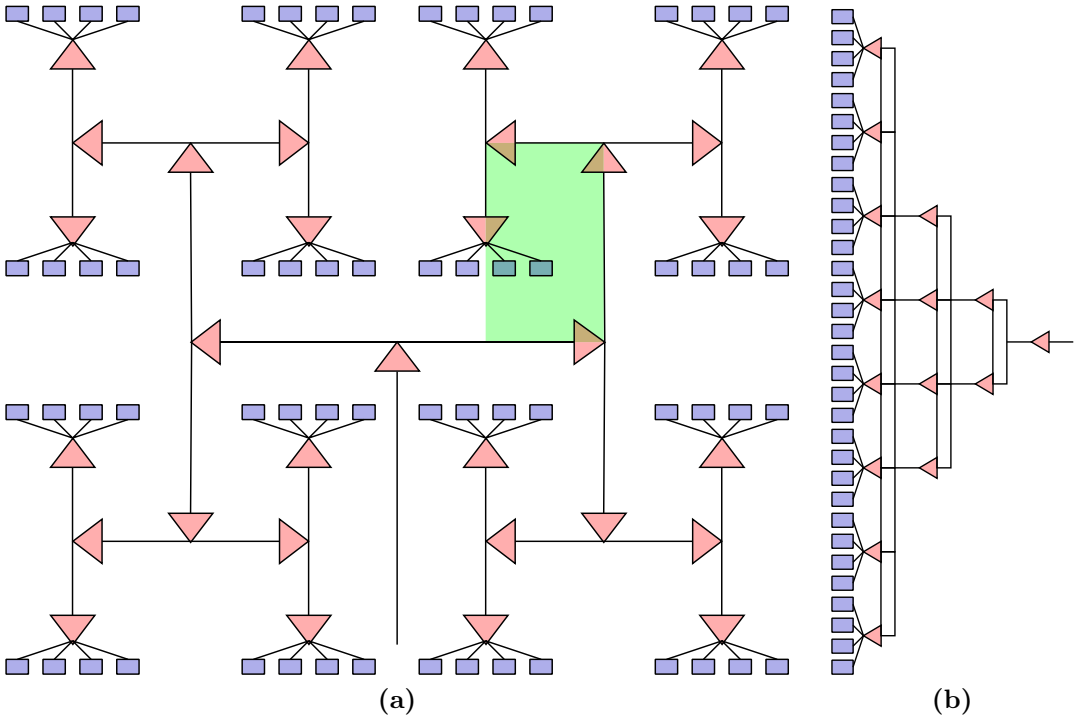


Figure 3.1: H-tree and Fishbone tree architectures. One the left the H-tree clock distribution. On the right the Fishbone clock tree. All the blocks in blue are cells to which the clock is delivered. On green, the area in the H-tree that can be used as a block in a hierarchical design. As clock tree levels are increased, that area becomes smaller, making it more difficult to come up with modular designs.

Even if terminations were applied at the two ends of each intermediate net to reduce the effect of reflections, the skew problem wouldn't be solved.

A new clock tree alternative will then be proposed, one that makes sure that the impedance seen at the output of each of the drivers is exactly the same for all the drivers in the same clock tree level. The proposed architecture will be based on the shape of an inverted cone. The exploitation of symmetry in this cone-shaped clock tree will allow to reach ultra-low skew.

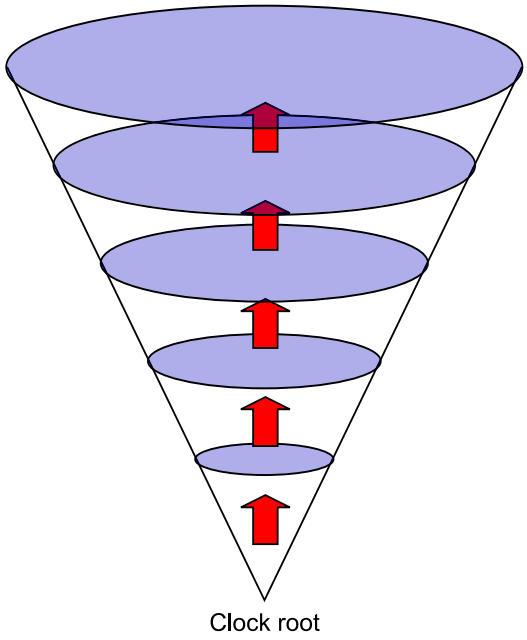


Figure 3.2: Inverted cone shape, inspiration for the *Conical-Fishbone* tree. Inverted cone shape used as inspiration for the design of a new clock tree architecture. Every circle gets excited by the circle below it. The clock root is the tip of the cone.

3.2 The Conical-Fishbone Clock Tree

The architecture proposed is based on the shape of an inverted cone, as seen in Figure 3.2. If several cross sections are created of the inverted cone, and each of these resulting rings is considered to be one of the many nets in a Fishbone clock tree, it can be seen that if a ring is excited evenly from the ring below, the circular characteristics of the wire will make the effect of reflections be exactly the same along any place in the wire. This idea is the one that will allow to achieve ultra-low skew.

Inspired on the inverted shape of a cone, in Figure 3.3 a new clock architecture is presented, one which will be called *Conical-Fishbone* tree. The resemblance to an inverted cone and to the Fishbone clock tree is easily seen. The clock root is

excited, and four equidistant places are used to excite the first ring of the tree. If the diameter of *RING 1* is x, then every time a clock tree level is added, the resulting ring increases by x. There is a linear relationship between the span of the tree and the number of levels in the tree. The following *RING 2* will be now excited in eight different equidistant places, but in order to maintain symmetry and equivalent load on each point where a ring is excited, two additional drivers are added, the ones in blue. These drivers have their output floating, they are just used to equalize the load along every ring. As the number of clock tree levels increases, the number of active buffers used to excite the following ring is $2.Ring\_n + 2$, where $Ring\_n$ is the ring number. When drawing the layout for this clock tree, the distances from the ring to the input of the exciting drivers from the same tree level were made sure to be exactly the same for all of them. It can be seen that all of the points in each of the rings where the ring is excited and/or read, see exactly the same impedance. This is the main characteristic that allow to achieve ultra-low skew.

Figure 3.4 presents a view of a CMP where the usage of the *Conical-Fishbone* trees can be seen. For both cases of the 64 and 128 PUs CMPs, the layout presented in Figure 3.4 is exactly the same. For both networks on chip, the clock frequency used will be 300MHz, and this clock will be distributed from a vertical clock tree cell, to each of the clock tree cells present for every row in the network. Along with the clock, an asynchronous global reset signal needs to be distributed to the networks on chip (NoCs) as well. This reset signal will also use the same structure as the before

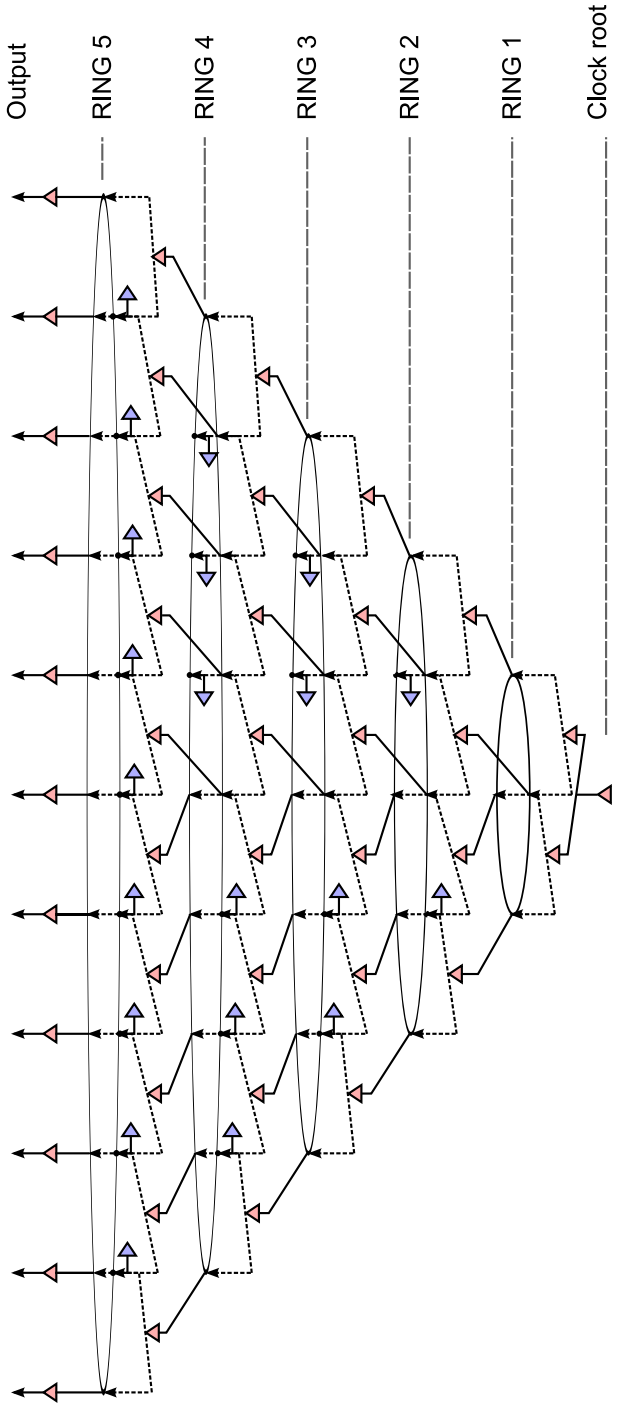


Figure 3.3: The *Conical-Fishbone* clock tree. A diagram of how this new architecture looks like is presented. All the active drivers from the same tree level experience the same output impedance, as well as the same input impedance. Floating output drivers (the ones in blue) are added to equalize the capacitance on every line.

mentioned clock. With respect to the *DDR DRAM PHY* block, this block uses four different clocks. A $1.25GHz$ clock ($0.8ns$ period) is used to send data from the *DDR DRAM PHY* block to the DDR memory. This clock is also sent to the 3D DDR chip. Another used clock for the *DDR DRAM PHY* is the previously mentioned clock, divided down to a $2.4ns$ period clock. These two clocks need to be in phase, and it is for this reason that only one clock tree cell is used for these two clocks, the division is done local to the output of the clock tree cell. The DDR memory sends back to the CMPs the clock that should be used to read the received data. This clock is also $1.25GHz$ and just like in the previous case, it is also divided down to a $2.4ns$ period clock. Both pair of clocks need to be available on both sides of the clock cell, it is for this reason that the output from one side of the clock tree cell is routed to the middle of the cell and then split in two.

Four clocks are needed in the *DDR DRAM PHY*, and two of them can be generated by the other two. In order not to multiply the area used by the *Conical-Fishbone* clock tree cells by two, the clock divided versions are generated locally to the output of the high frequency clock tree cell. This allows a reduction by half of the area used for the clock trees in the *DDR DRAM PHY* block. Figure 3.5 shows the augmented clock tree cells used in the *DDR DRAM PHY*. Along with the two clocks, a clock divider and a buffer can be found. The clock divider runs a counter that counts from 0 to 2, and the buffer is placed to compensate for the delay introduced by the clock divider. The reset signal is used to reset the clock dividers to a default state so that

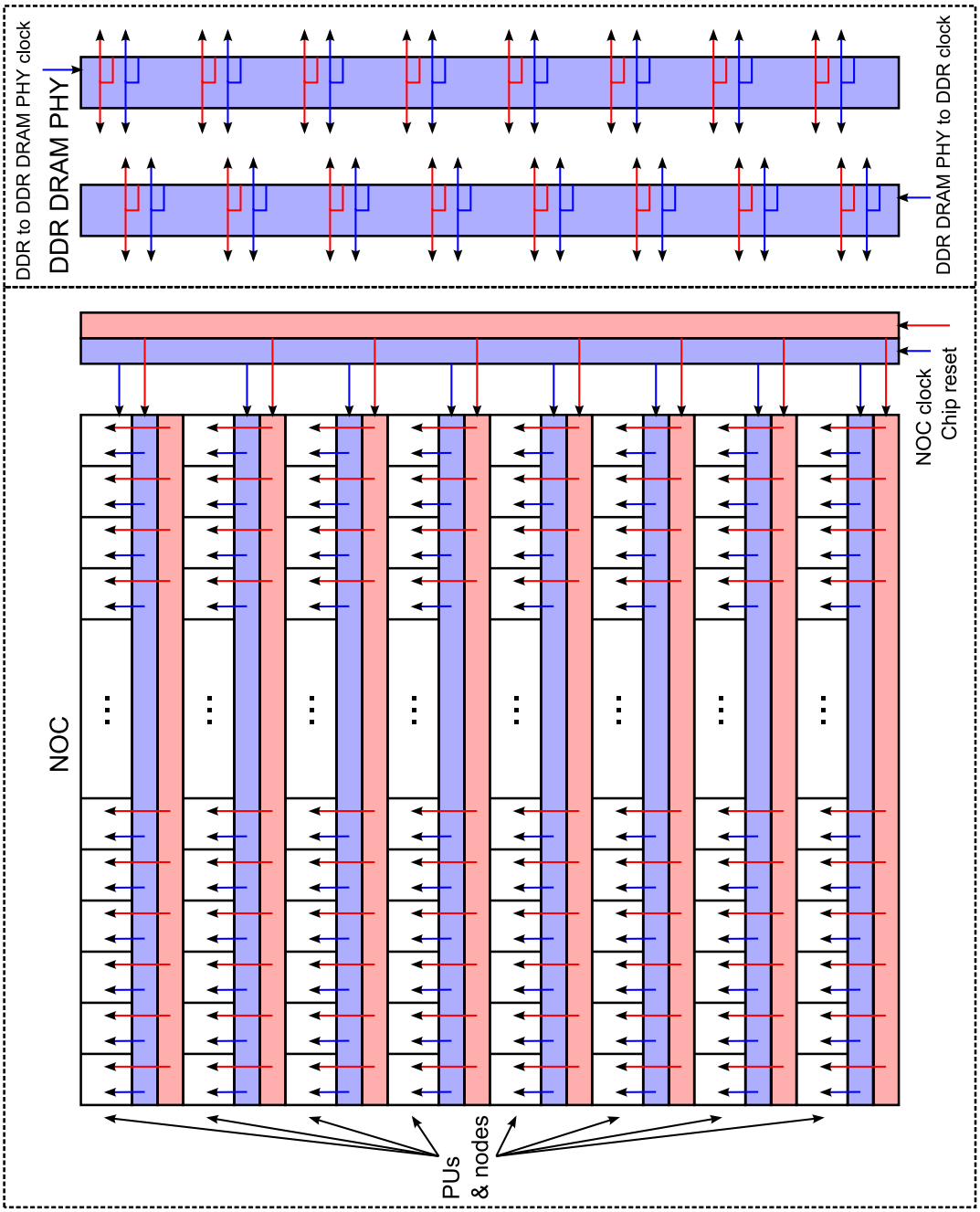


Figure 3.4: Implementation of the *Conical-Fishbone* tree in the CMPs. An outline of where the different *Conical-Fishbone* clock trees are used. On the NoC side, the blue cells are the ones delivering the $300MHz$ clock, and the red cells are delivering the global reset signal. On the *DDR DRAM PHY* side two clock tree cells are used for the clock used to send data to the DDR memory and for the clock used to read the data back. The clock periods used for these cells in the *DDR DRAM PHY* side are $0.8ns$ and $2.4ns$ (approximately $1.25GHz$ and $420MHz$).

all of the clock outputs can be in phase. Because the clock divider counter has a period of 3, from one clock output to the next one, the reset signal is registered three times. This will ensure that after a certain number of clock cycles, all of the clock tree outputs will be completely in phase.

The size of each of the clock tree cells for the *DDR DRAM PHY* is approximately $13.44mm$ by $50\mu m$. Each of the long sides has 64 clock outputs for both fast and divided clocks. Figure 3.6 shows the outputs for the fast clock in the *DDR DRAM PHY* block clock cells. It can be observed that for a distance of $13.44mm$ the maximum skew achieved is just $31.8ps$. The results obtained in this figure consider all of the capacitance and resistance parasitics extracted from the layout of these cells.

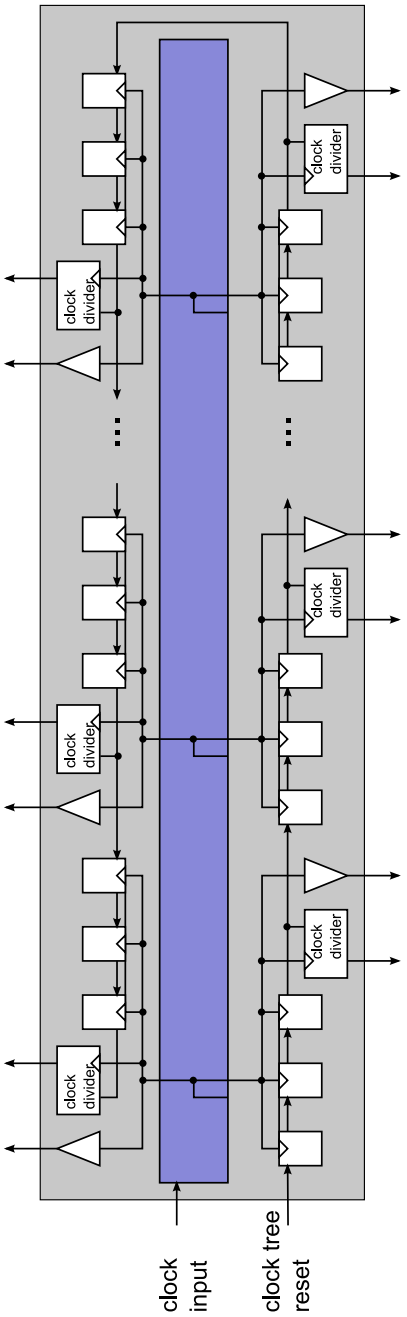


Figure 3.5: Clock tree cell used in the *DDR DRAM PHY*. An augmented version of the clock tree cell used for the *DDR DRAM PHY* is shown. This clock tree cell receives a $0.8ns$ clock signal and it outputs two frequency clocks, $0.8ns$ and $2.4ns$ clocks. The generated $2.4ns$ clock is created by using a local counter to each clock output that divides by three the $1.25GHz$ clock frequency. All of the $2.4ns$ clock outputs are in phase due to the usage of a reset signal which arrives to all of the clock dividers at multiples of three clock cycles.

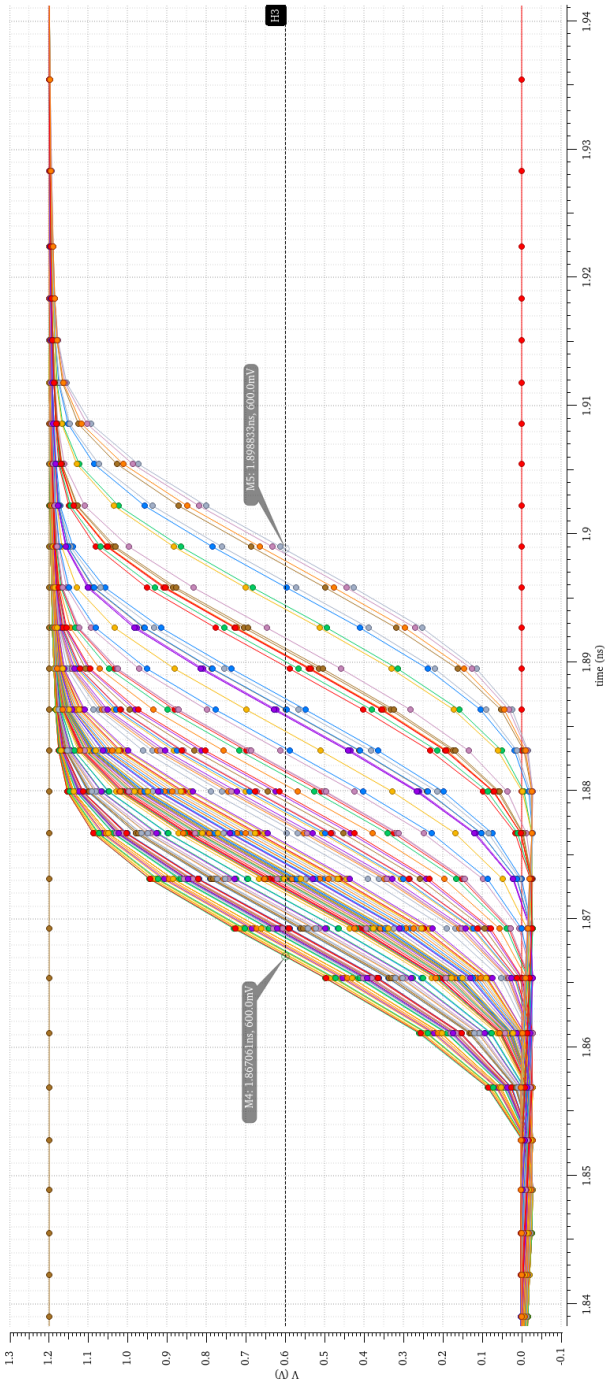


Figure 3.6: *Conical-Fishbone* skew simulation. Simulated outputs of the $0.8ns$ clock propagated through the clock cell in the *DDR DRAM PHY*. For all of the clock outputs along both sides of the cell, extending for $13.44mm$, only $31.8ps$ of skew was found. This simulation has been done considering all the capacitance and resistance parasitics of the clock tree layout.

Chapter 4

NoCs

4.1 First Level NoC Architecture

Two network levels were introduced in the previous section. One of those networks supports the communication among all of the PUs (*L2 network*), and the other provides access to the 3D-DiRAM for every single PU (*L1 Network*). The focus of this section will be on the later network.

Several options were analyzed for the communication of the different PUs to the 3D-DiRAM, but the main characteristic that made the choice of a token-ring like network, was the necessity of a warrantied throughput for every PU. Warrantied throughput is a very difficult thing to achieve is one wants flexibility in a network. An allotted network slot would need to be given to every PU as a possible solution to warranty a minimum throughput. This scheme would look like a superposition

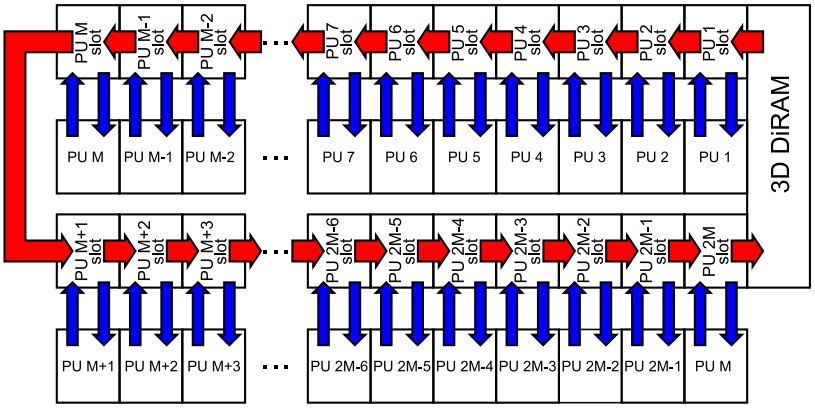


Figure 4.1: Token ring network approach for the *L1 network*. This figure shows the token-ring approach used for communicating all of the PUs with the 3D-DiRAM. Every PU has network slot assigned with its own address.

of several networks, where each one acknowledges only one PU. With this kind of predictability for this network, not any network shape is easy to implement. The allotted slot for each PU would probably need to travel through a known path in the network, making many network shapes, such as meshes, not suitable for this problem. It is for this reason that a ring was found to be the best shape. All the packets in the network flow with the same pattern, making the sharing of throughput among all PUs much simpler. In Figure 4.1 the ring network with the assignment of dedicated slots for each PU is shown. If the network slots were not assigned to a particular PU, a PU could make use of all the slots, leaving the remaining PUs waiting for a free slot that may never come.

The shape for the *L1 network* has been defined as a ring, but the question arising now is if a single ring should be used for all of the PUs on a CMP or not. For the traffic between the *DDR DRAM PHY* and the *L1 network*, is it better one or more

connections in the distribution of packet traffic? If only one ring was used, then the summation of the delay a packet suffers from a PU to the *DDR DRAM PHY* and vice-versa would be constant, but high. For the two networks shown in Figure 2.6 and 2.7 this delay summation would be 64 and 128 respectively. This delay was considered too high, and on top of that, if any of the *L1 network* nodes didn't work because of fabrication problems, then none of the PUs would have access to the DDR memory. Consequently several token-ring networks were designed, one per each PU row, as seen in Figure 2.6 and 2.7, making a total of eight token-ring networks. This design now is more fault-tolerant, as a faulty node would only disable one of the ring networks. For the 64 and 128 PUs CMPs, the delay summation would now be 16 and 32 respectively, which is a much more reasonable delay. The delay is not 8 and 16 as one would expect, which is the number of PUs on each row of the CMPs, because with a ring for each row, there will be a path going right and another one going left, making up to 16 and 32 available slots per each token-ring network. Figure 4.2 shows the token-ring network present in each of the 64 and 128 PUs CMPs rows. The places were the different PUs tap into the network have been distributed equidistantly so that a more uniform delay could be achieved on each of the rows. As it will be later explained, the *DDR DRAM PHY* will accumulate the requests from the PUs of a certain row in a buffer local to each row. It is after this local buffer has been filled, or a certain amount of inactivity time has been sensed, that the read and write commands are performed by the *DDR DRAM PHY*. Consequently, when the DDR

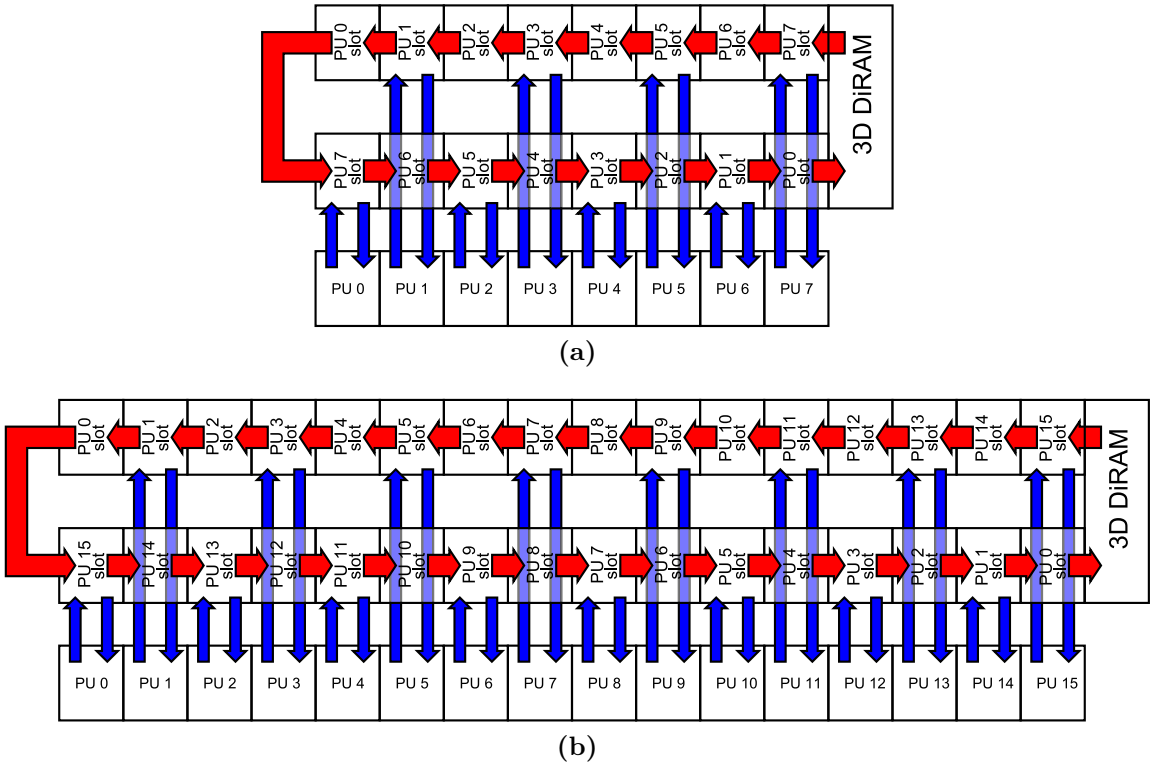


Figure 4.2: *L1 network* token-ring per row. On the top the token-ring network implemented for the 64 PUs CMPs and on the bottom the one for the 128 PUs CMPs. 16 and 32 are the available slots for each of the rings.

memory access time is considered, this additional delay will have to be taken into account.

Modularity was exploited as much as possible for these designs (see section 2.1). For the case of the *L1 network*, two different types of network node modules were designed that allowed the equidistant tapping of the PUs onto the token-ring networks. The connectivity between the ring network and the PU is represented in Figure 4.3. These two types of nodes were placed along each row alternating between the two of them.

Each of the nodes on a token-ring possesses its own address local to that ring.

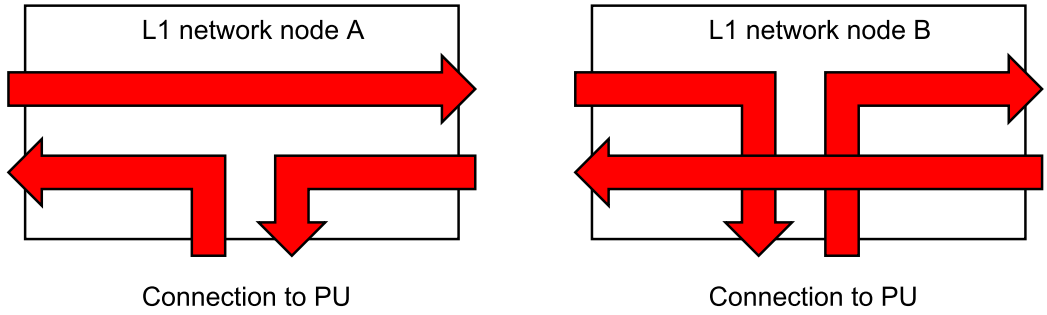


Figure 4.3: Two types of *L1 network* nodes. Two types of *L1 network* nodes are presented here. These two types of nodes allowed the equidistant tapping of the PU into the token-ring networks.

So that the same node could be used for both 64 PUs and 128 PUs CMPs, the number of bits used for the local address in both cases is the same. Four bits are required to differentiate each of the PUs in a ring. Additional three bits are added so that the *DDR DRAM PHY* can tell to which token-ring packets should be sent. A total of seven bits are necessary to identify where a packet should be sent from the DDR memory. As it will be discussed later, because read and write commands are not warrantied to be executed in the received order by the *DDR DRAM PHY*, an additional field was added along with the packet sent into the *L1 network*. This field is a tag field that allows the sender to identify the transaction sent uniquely.

4.2 Second Level NoC Architecture

4.2.1 Introduction

The architecture for the second level NoC present in the CMPs is here introduced. This network is called the *L2 network*, and will be the one communicating all of the different PUs on chip. Some of the desired characteristics for this network are:

1. Free from dropping packets.
2. Reduced usage of resources.
3. Multicast routing.
4. Finite latency for a packet to get delivered, meaning that no dead-locks or infinite loops (live-locks) for packets are allowed.

With the first point satisfied, packets in the network will never vanish unless each packet reaches its destination. This can be achieved by making sure that everything that comes into a node can be forwarded to its outputs. In a node, this can be accomplished by having the same number of inputs as outputs (no sinks). The number of inputs and outputs in a node will be M. In making this claim the possibility of suffering incorrect routing of up to $M - 1$ packets in a node has to be taken into account, since at least one packet should be routed correctly. Networks that will always forward its inputs to its outputs will then be considered, making this network

contain packets that will never remain still. For this particular type of network, processing units will be located in each of the routing nodes. These units will inject packets to the network as long as at least one of the node inputs is free. The interface from the processing unit to the node can be considered an additional input to the node, but this input does not come into play unless one of the additional M inputs is free. This means that even in the case in which all of the processing units in all of the nodes have data to send, and they are waiting for an available slot, the routing of the packets already in the network will not be affected.

The second point is very much related to the first one. If packets are allowed to be routed incorrectly, then there is no need for buffering any input of a node. Many approaches to networks, such as meshes, rely on FIFOs that are placed in each of the inputs of a node to deal with packets that need to be routed to the same output. The size of these FIFOs can be determined using queuing theory, so that bursts of packets coming into a node that need to be routed to the same interface, can be dealt with. Already the choice of a FIFO size is problematic, because it is tailored to the traffic expected in the network. This gives room to cases in which these FIFOs will not be big enough, and packets will then be dropped. With the approach taken for this project, not only packets are not dropped, but input FIFOs are not needed, making the amount of silicon area used for the routing network on chip decrease significantly.

The third point is very difficult to satisfy considering the previous desired characteristics. A node might require to send a multi-cast packet to several outputs, and

then either buffering is required, or the number of outputs in a node needs to be greater than the number of inputs. This is because implicitly a multi-cast packet represents several input packets. It is for this reason that multi-cast routing will not be considered for this network. Multi-cast will be addressed by sending several uni-cast packets.

The final point is the most difficult one to achieve, and will be the focus of the next section. Some previous work related to the problem desired to solve has been proposed in,[19] where the author points out some of the key features of different types of networks, such as Centralized, Decentralized and Distributed networks that helped to understand the pros and cons of the types of networks to consider when designing a NoC. In the case proposed the number of connections for each node is of high importance, because when building NoCs, wire buses take a huge amount of area on chip, and if one wants to benefit from the topology of the network when laying down the network nodes on silicon, then the most appropriate network would seem to be the distributed one.

The concept of incorrect routing is called *deflection routing* and the idea of eliminating FIFOs at the nodes' inputs and outputs (*buffer-less routing*) has not been addressed much, but in just a few cases that are now presented. In[20] the author proposes two algorithms for performing routing in two specific network topologies, a nxn tourus and a $N = 2^n$ nodes hypercube (n dimension hypercube). For this two architectures the author obtains $2n + O(log(n))$ and $O(n)$ steps for the delivery of a

packet with high probability. Again, the main striking feature in this work is that there are no buffers or FIFOs at intermediate nodes, so packets may be sometimes routed in the wrong direction. In the case of the network presented in this work, the idea of buffer-less and deflection routing will be used, meaning that some of the packets will be routed incorrectly, but it will be shown that the proposed algorithm is not only valid for the specific network topologies like Hypercubes or Touruses, but for any network topology that follow certain rules.

In[21] the importance of buffer-less routing is introduced as a very attractive and energy-efficient design option for on-chip cache/processor-to-cache networks. The author presents the idea of ranking rules, which allow to decide how to route the packets arriving to a node that desire to be routed through the same port. A combination of *oldest first* (OF) priority plus a type of port prioritization on each node is proposed, and the author claims that eventually all the packets in the network will be delivered without dead-locks or live-locks. This approach is very interesting, but the author fails to provide an expression that characterizes the delay a packet suffers in the network due to deflection or how to control the number of times a packet has been deflected, because this second level of priority proposed (*port priority*) is not based on any network activity history, which will be shown that in the case of this work, it does (increasing the throughput of the network). A better approach would be the combination of OF and an indicator of the number of times a packet was deflected in order to take a more informed and efficient decision of how a packet should be routed.

An alternative algorithm will be proposed for routing packets that takes into account OF and the number of times a packet has been deflected allowing the delivery time to suffer less variation, allowing network traffic to be more uniform.

At last an additional approach was taken in[22] where instead of having the two levels of priority (OF and port priority from[21]), deflection is only controlled for what the author calls *Golden Packet*. There is only one *Golden packet* at every single time step in the network, and this packet has the highest priority, so this packet will go directly to where it is supposed to. After being delivered, another packet will be designed as the *Golden packet*, and so on and so forth. By induction it can be seen that every single packet will be delivered with no dead-locks or live-locks, and the logic for performing the routing is very simple (less silicon area), but the trade-off is that the delay for a packet to be delivered increases significantly, making real-time applications not a good match for this type of routing network.

4.2.2 The Proposed Network Solution

A total of M inputs and M outputs will be present in each of the nodes for the network presented in this work. The maximum number of packets that can be allocated in the network will be N, which is the total number of unidirectional edges in the network graph. The last assumption made for this network is that, at each node, the routing tables are such that there is always a feasible path from node i to node j where $i \neq j$. A way of performing routing on the network will be

presented, ensuring that all of the packets in the network get delivered in a finite amount of time, without live-locks or dead-locks. Knowing all of the packets will be delivered, and having a very flexible condition for the routing tables in the network, reprogramming of routing tables can be performed if desired, allowing to distribute traffic more uniformly depending on the task at hand. The proposed solution as it will be seen, is not fixed to any type of network topology as it is the case of all the cited work in 4.2.1, any topology such as a tourus, mesh, etc, can be used as long as the number of input and outputs in a node are the same, and a feasible routing path can be found from any node to any node of the network.

In Figure 4.4 an example of how packets are generated in the network is provided. For this example network, $M = 4$. With every time step, new packets can be injected to the network. A counter tc is added to each packet, starting with a count of 0. This is the *time counter*, which records the amount of time a packet has been circulating in the network. This counter is increased by one with every node hop. It is reasonable to think that packets that have been traveling for longer time should have more priority of being correctly routed than those that were just injected into the network. It is for this reason that this counter will be used as the priority a packet has to be routed correctly. If two or more packets arrive to a node, and one of them has a highest and unique priority, then that packet should be routed correctly. The remaining ones might or might not be routed correctly, but the highest priority one will always be.

At any given time, the network will hold a counter value that will be the highest.

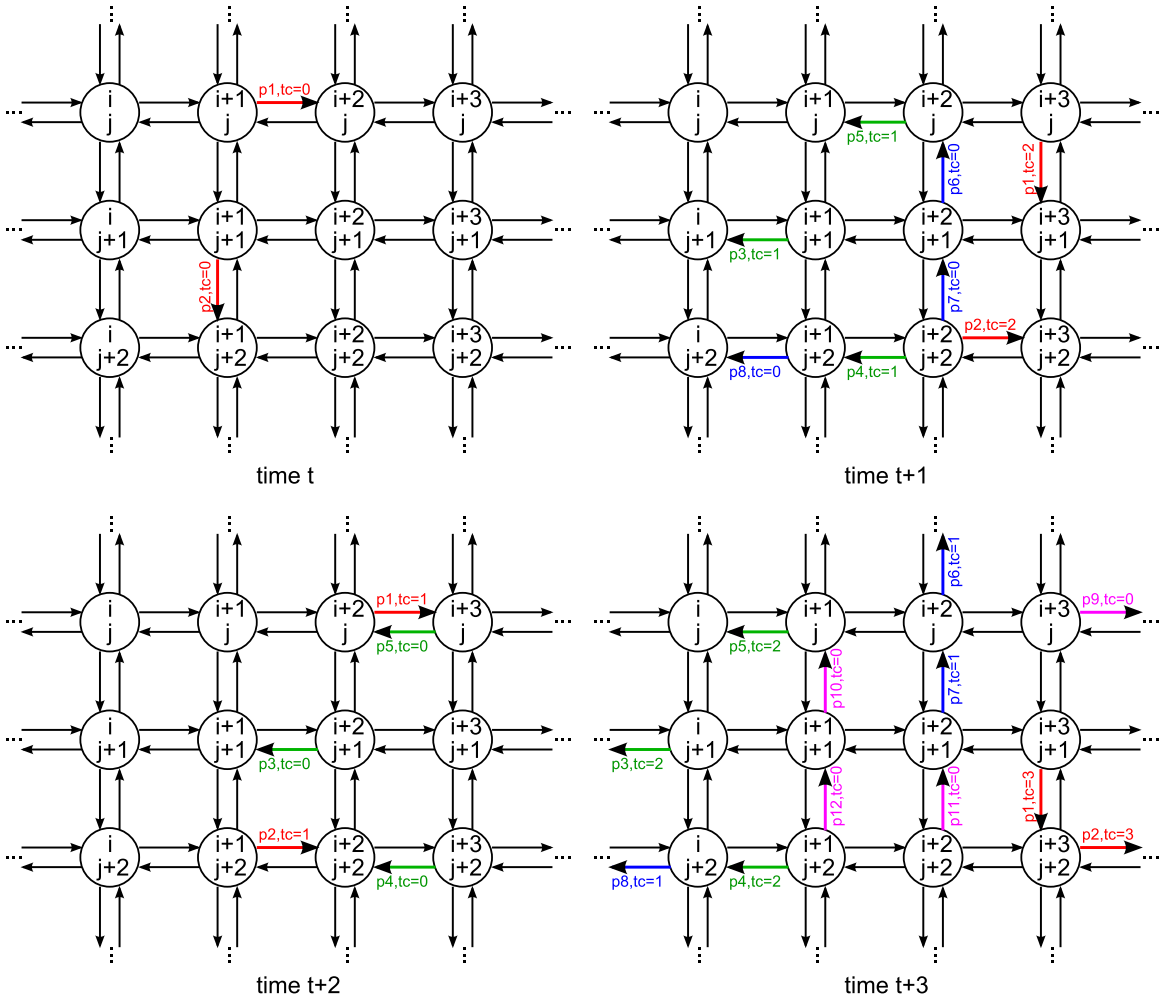


Figure 4.4: *Time counter* evolution example. Example of how packets are routed in a network with $M = 4$. Packets are generated in this case with every new time step. A *time counter* is attached to each routed packet, where each node hop increases its count by one.

The set of packets with this value can go from 1 to N. For the example network in Figure 4.4, the highest priority set of packets is $HP = \{p1, p2\}$. For this set of packets, all the other packets in the network have less priority than them, and this will always be the case, until these packets get delivered and a new set of packets will take the place of the highest priority ones. Consequently, the highest priority set of packets HP will never notice the presence of the lower priority ones, they will only notice the ones from their own HP set.

Now, if a way of making sure that all of the packets in set HP can be routed to their destination in a finite number of node hops can be found, the moment all of the packets in set HP get delivered, set HP will be updated with the packets with the highest priority at that time. The same argument can now be applied to this updated set HP. By induction it can be ensured that every single packet injected into the network will be routed to its destination in a finite amount of time.

With the addition of the *time counter*, packets generated at the same time will have the same priority, and if they happen to be the oldest ones in the network, then they will be the most favored ones when routing them to their destination. When a fight among two or more packets happens in a node, if all of the packets contain a different *time counter* value, there won't be any confusion in how routing should be performed. The problem arises when packets with the same *time counter* need to fight to get routed correctly. It is for this reason that a second level of priority, internal to the set of packets with the same *time counter* could solve this problem.

If somehow all of the second level priorities of a given set of packets with the same *time counter* could be all different, then no confusion will arise when trying to route the packets through the network. With all different priorities, in the whole network, there will always be a packet with the highest priority of all, and that one will be sent to its destination with no mis-routes. A new counter will now be introduced, the *fractional counter*. This counter will be the one that will help in the process that diversifies the second level priorities defined by the *fractional counter*. Every packet will now have two counters, the *time counter* and the *fractional counter*.

With the *fractional counter*, fights in a node will not happen when the *time counter* of the packets are the same, they will happen when additionally their *fractional counter* are equal as well. In explaining the mechanism used for updating the *fractional counter*, Figure 4.5 is presented. Two or more packets with the same *time counter* values arrive to a node. This set will be called $\vec{X_1}$. Additionally, one or more packets with a higher *time counter* arrive to the node as well, this set will be called $\vec{X_2}$. All the packets in set $\vec{X_1}$ and $\vec{X_2}$ need to be forwarded to the same node output interface. Only one of the packets in the highest priority set $\vec{X_2}$ will be routed to the desired output, the other ones will be routed incorrectly. For the case of set $\vec{X_1}$, all the packets will be routed incorrectly. Two possible scenarios are then possible when fighting, the set of packets fighting all get routed incorrectly, or if this set of packets is the one with the highest *time counter* in the node, one of the packets is correctly routed. From Figure 4.5 it can be seen that for both $\vec{X_1}$ and $\vec{X_2}$, their *fractional*

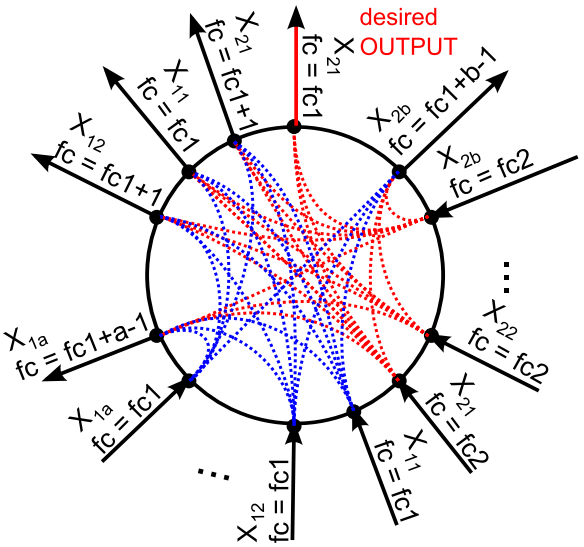


Figure 4.5: ***Fractional counter*** **update example.** In this node, inputs in vector $\vec{X_1} = [X_{11}, X_{12}, ..., X_{1a}]$ will have the same priority, in this case the same *time counter* and *fractional counter* $fc = fc1$. Additionally the inputs in vector $\vec{X_2} = [X_{21}, X_{22}, ..., X_{2a}]$ will also have the same priority, the *time counter* will be the same for all of the elements in $\vec{X_2}$, but higher than the elements in $\vec{X_1}$. The *fractional counter* for all the elements of $\vec{X_2}$ will be $fc = fc2$. All of the packets want to go to the *desired OUTPUT*. Only one input from $\vec{X_2}$ will be redirected correctly, in this case X_{21}, everybody else will be routed incorrectly. For the case of $\vec{X_1}$, all of its elements will be routed incorrectly

counters are diversified to all different values. Next time any of the packets in both vectors find any of their vector mates, no confusion will arise regarding who has the highest priority.

In Figure 4.6, the evolution of the *fractional counter* depending on the number of packets fighting in a node is shown. After a time step the *fractional counters* get diversified to up to M different values. Remember that staying in the same *fractional counter* does not mean that the routing was done correctly, unless that packet belongs to the set of packets arriving to the node with the highest *time counter*.

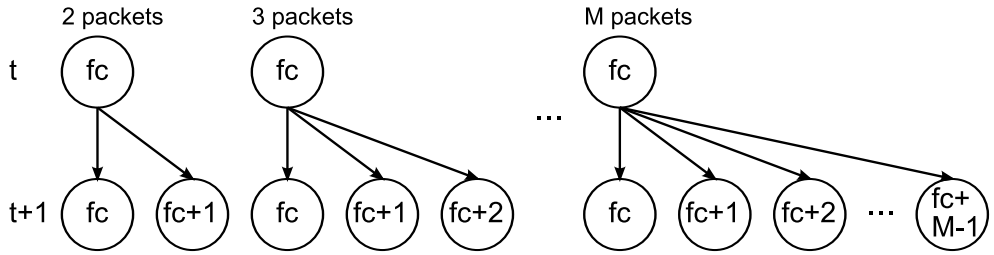


Figure 4.6: *Fractional counter* update values. Evolution of the *fractional counter* in one time step depending on the number of packets fighting in a node.

For the case of the set of packets with the same *time counter*, that is not the highest in the network, the evolution of all its packets can be expressed using the graph shown in Figure 4.7. They cannot be the set with the highest *time counter* because when becoming the set with the highest *time counter*, all the packets will not necessarily have their *fractional counter* starting at 0. The coefficients from vector $\vec{D(t)} = [D_0(t); D_1(t); D_2(t); ...; D_{t(M-1)}(t)]$ in Figure 4.7 represent the distribution of packets among all possible values the *fractional counter* can achieve at time t. Because with each time step an additional $M-1$ possible states for the *fractional counter* are added, then the total possible states for this counter at time t is $tM-(t-1)$. Because the maximum number of packets the network can hold is N, it can then be said:

$$\sum_{i=0}^{t(M-1)} D_i(t) \leq N, \forall t \geq 0 \tag{4.1}$$

As packets evolve going down the graph, $\sum_{i=0}^{t(M-1)} D_i(t)$ can stay the same or decrease over time, but never increase. This is because packets can reach their desti-

nation as they go down. This can be expressed mathematically:

$$N \geq \sum_{i=0}^{t(M-1)} D_i(t) \geq \sum_{i=0}^{(t+1)(M-1)} D_i(t+1), \forall t \geq 0 \tag{4.2}$$

Let's now analyze if there is a maximum count the *fractional counter* can reach for the non-maximum *time counter* set in the network. If there is a maximum, let's consider it achieved at time t'. At that time, $Mt' - (t' - 1)$ are the possible bins packets can be distributed in. To analyze the maximum count the *fractional counter* can achieve, the most right bin at time t' needs to be considered, taking as a reference the graph in Figure 4.7. If the maximum value coefficient $D_{t'(M-1)}(t = t')$ can achieve could be found to be less than one, then that bin will never be populated, and then the *fractional counter* can only count so far. For finding the maximum value this coefficient can achieve, the most possible packets need to be moved to the right of the graph as they evolve in time. In Figure 4.6 the way the *fractional counter* gets diversified depending on the number of packets fighting in a node was presented. In order to achieve the most number of packets in the most right bin in any of the transitions in Figure 4.6, fights should be maximized. If for some reason a set of packets does not fight, then they will not add any count to the most right bin. The distributions that maximize the count in the most right bin in a unit time step, depending on the number of packets fighting, are presented in Figure 4.8. These distributions are different depending on which of the two previously mentioned cases

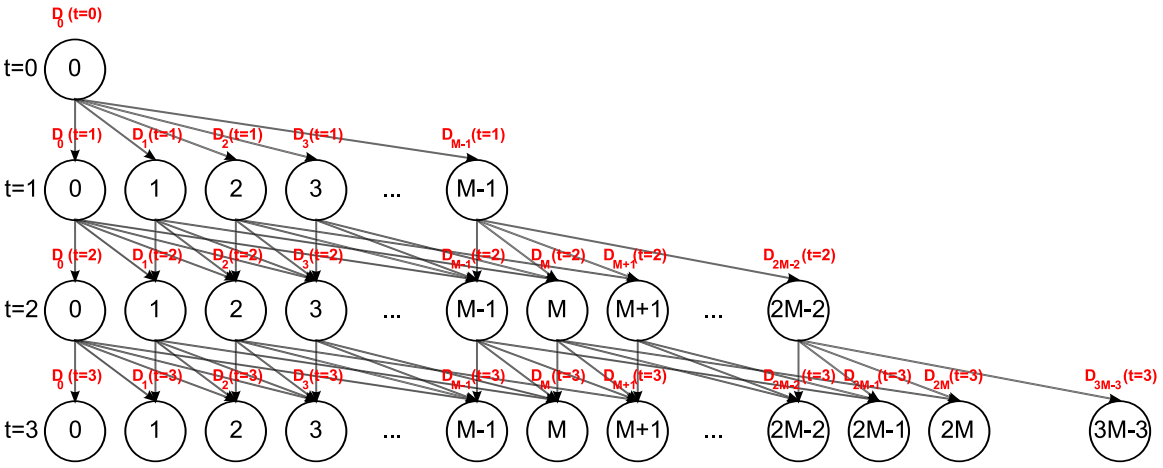


Figure 4.7: Evolution of the distribution of packets with the same *time counter* over time. These packets are not considered the set with the highest *time counter*. It cannot be the set HP because that set will not necessarily start with all of its packets in bin 0.

one is. These cases are the ones in which one packet gets delivered correctly and the other one is when none of them are.

From Figure 4.8 two possible cases can be seen for the sets of packets without the highest *time counter*. For the one on the left always from a fight one packet gets routed correctly, but on the one on the right all of the packets are routed incorrectly. Let's consider only the case that gives the maximum count to the most right bin. Let's assume that the case on the left from Figure 4.8 is the one that gives the maximum count to the most right bin. Then:

$$K\frac{1-\frac{1}{M}}{x} \geq K\frac{1}{x+1}, x \in [1; M-1]$$

$$(x+1)(1-\frac{1}{M}) \geq x$$

$$(x+1)(M-1) \geq Mx$$

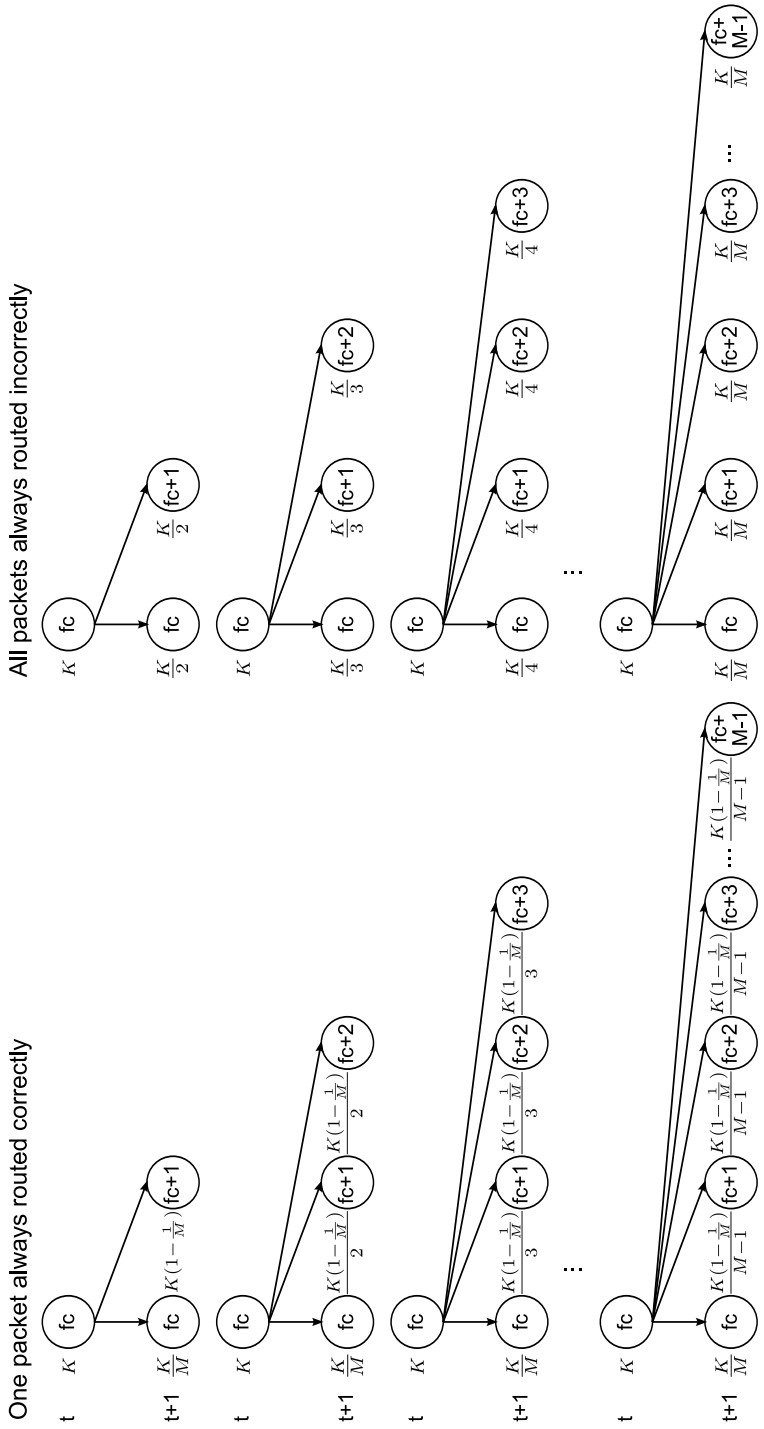


Figure 4.8: *Fractional counter* update when trying to achieve its highest value. For a diversification of the *fractional counter* of up to M different values, the distribution that maximizes the count in the most right bin, depending on the number of packets fighting, is presented. K is the number of packets, $K \leq N$.

$$xM + M - x - 1 \geq Mx$$

$$\Rightarrow M - 1 \geq x \tag{4.3}$$

The result in 4.3 is then correct, and then the case on the left from Figure 4.8 is the one that rises the maximum count to the most right bin.

Let's now consider vector $\vec{Y}$ a vector that expresses how many shifts to the right are taken for every time step to get to a certain bin. A zero in this vector would mean that no shift was made to the right, and then it can be assumed that up to all of the packets can stay in that bin. Consequently a 0 will not necessarily decrease the number of packets along the way. Vectors in which all of the elements are different than zero will then be considered. An example considering $M = 4$ is shown in Figure 4.9. Let's consider $\vec{Y'} = [2; 3; 1]$. This vector will make a packet go to bin $\sum_{i=0}^{L(\vec{Y'})-1} = 6$. Many vectors can make a packet go the same bin, the elements in $\vec{Y'}$ need to just be permuted, and all of these paths will give rise to the same maximum number of packets for the target bin. The reason why this happens is that as one goes down the graph, multiplication is performed, and multiplication has the property of being commutative. All of the possible paths for the chosen vector can be seen in different colors.

Let's consider now two vectors $\vec{Y_1} = [y_0; y_1; y_2; ...; y_L]$ and $\vec{Y_2} = [y_0; y_1; y_2; ...; y_{L1}; y_{L2}]$. The condition is that $y_L = y_{L1} + y_{L2}$, and this means that both vectors go to the same destination, but $\vec{Y_2}$ takes an additional time step. Breaking the vector $\vec{Y}$ will

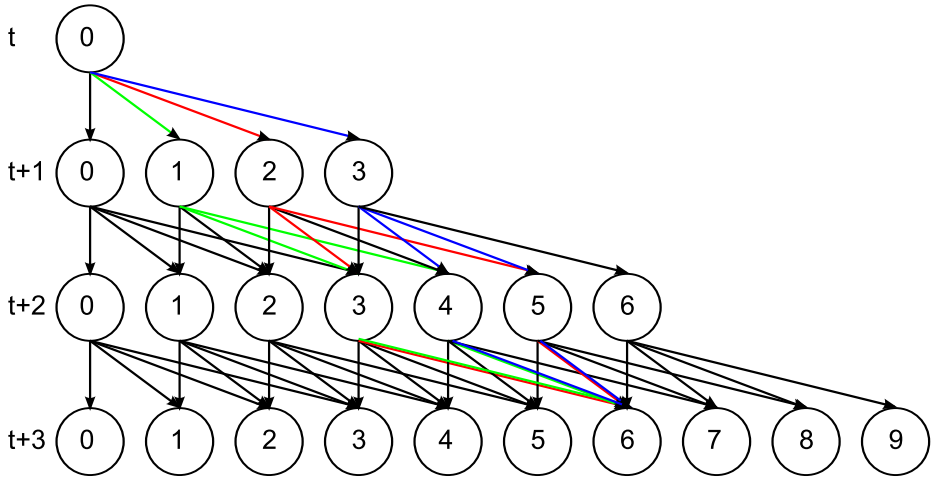


Figure 4.9: Different paths arriving to the same bin. All possible paths found for the case $\vec{Y} = [2; 3; 1]$. These are 6 different paths.

be demonstrated to diminish the maximum number of packets at the target bin at any time t. Let's make that assumption:

$$\prod_{i=0}^{Length(\vec{Y_1})-1} \frac{1}{y1i+1} \geq \prod_{i=0}^{Length(\vec{Y_2})-1} \frac{1}{y2i+1} \tag{4.4}$$

$$(\prod_{i=0}^{Length(\vec{Y_1})-2} \frac{1}{y1i+1})\frac{1}{y_L+1} \geq (\prod_{i=0}^{Length(\vec{Y_2})-3} \frac{1}{y2i+1})\frac{1}{y_{L1}+1}\frac{1}{y_{L2}+1} \tag{4.5}$$

$$(\prod_{i=0}^{Length(\vec{Y_1})-2} \frac{1}{yi+1})\frac{1}{y_L+1} \geq (\prod_{i=0}^{Length(\vec{Y_1})-2} \frac{1}{yi+1})\frac{1}{y_{L1}+1}\frac{1}{y_{L2}+1} \tag{4.6}$$

$$\frac{1}{y_L+1} \geq \frac{1}{y_{L1}+1}\frac{1}{y_{L2}+1} \tag{4.7}$$

$$\frac{1}{y_{L1}+y_{L2}+1} \geq \frac{1}{y_{L1}+1}\frac{1}{Y_{y2}+1} \tag{4.8}$$

$$(y_{L1}+1)(y_{L2}+1) \geq y_{L1}+y_{L2}+1 \tag{4.9}$$

$$y_{L1}y_{L2} \geq 0 \tag{4.10}$$

The result in Equation 4.10 tells that, if the maximum number of packets is desired in a target bin, then breaking $\vec{Y}$ into more time hops will just decrease the maximum number of packets at the destination bin. Consequently, in maximizing the number of packets in a target bin, one needs to minimize the time steps, making the biggest jumps to the right as possible. The maximum number of shifts to the right that can be taken in one-time step is $M-1$.

Let's now find if there is a maximum value for the *fractional counter*. At time $t=t'$ the number of possible bins is $t'M-(t'-1)$. The number of required jumps to the right to get to the most right bin at time t' is $t'(M-1)$. The maximum value for $D_{t'(M-1)}(t=t')$ is:

$$D_{t'(M-1)}(t=t') = \prod_{i=0}^{t'-1} \frac{(1-\frac{1}{M})}{M-1} = \frac{1}{M^{t'}} \tag{4.11}$$

If one starts with the maximum possible number of packets N, the maximum packets for the *fractional counter* $t'(M-1)$ is $N/M^{t'}$. In order to see if the *fractional counter* can reach a maximum value, one needs to make:

$$N/M^{t'} < 1 \Rightarrow t' > log_M(N) \tag{4.12}$$

Consequently now one can find the first t' that satisfies condition 4.12, and can then bound the *fractional counter* to be represented with a finite number of bits. Remember that conditioning $t' > log_M(N)$ will make *fractional counter* equal to

$t'(M-1)$ an impossible state to reach, not even for one packet. One can argue that other paths could be adding packets as well for that particular bin, but it has been demonstrated that the one that gives the most, does not even give one packet, and then none of the other paths will add any additional packets. For the next time step $t'+1$ the maximum number of packets in bin $t'(M-1)$ will be the same, but now $M-1$ additional bins to the right have been incorporated. If one can prove that those bins have even less chance of having any packet, then a proof has been shown. This prove is very simple. Any of the paths that give the maximum packets for bins $t'(M-1)+1$ to $(t'+1)(M-1)$ can be used for proving this. The proof is the following:

$$N(\prod_{i=0}^{t'(M-1)} \frac{1-\frac{1}{M}}{M-1})\frac{1-\frac{1}{M}}{k} < N\prod_{i=0}^{t'(M-1)} \frac{1-\frac{1}{M}}{M-1} = \frac{N}{M^{t'}}, \forall k \in \{1,2,...,M-1\} \quad (4.13)$$

It has been shown that for the sets of packets without the highest *time counter* in the network, the *fractional counter* cannot exceed $t'(M-1)$, where t' is the first t that satisfies $t' > log_M(N)$, where N is the maximum number of packets the network can hold, and M the number of inputs/outputs in each node.

Let's now analyze if for the case of the HP set this also happens. Figure 4.8 presents the two cases packets from sets that are not the ones with the highest *time counter* can experience. For maximizing the number of packets on the right most bin at any given time the case on the left was considered. For the case of the highest priority set HP, the case of the left in Figure 4.8 is the only one that can be possible,

because there is always a packet being routed correctly. Consequently, the previous analysis works for the HP set as well. One can additionally conclude that this analysis not only works for networks with M connections per node, but for networks where there is up to M connections per node. By having less connections than M, some of the nodes will experience that the maximum shift to the right they can provide is less than $M-1$. Therefore, the maximum number of packets in the most right bin at time t' will be even less compared to the case where all of the nodes have M connections. This statement will be particularly useful when designing networks were boundaries might have less node connections, like in the case of the boundaries of a mesh network.

4.2.3 Simulation Results

Networks with completely random connectivity, but satisfying the existence of a feasible path from any node to any node of the network have been simulated. The transition from any node to one of its neighbors can be seen as a Markov chain. The required existence of a feasible path from any node to any node of the network is equivalent to asking that the equivalent Markov chain transition matrix has to be irreducible. Let's consider $P \in \mathbb{R}^{NxN}$ a full rank transition matrix. For P, each column or row will have only M values different than 0, meaning that each node only connects to other M nodes. The values in the diagonal of P will be zero, because packets are delivered to the local node or forwarded to other nodes. To make things

simpler it can be assumed that all of the values different than zero in matrix P to be $1/M$. This would mean that the transition to neighbors is equiprobable. If starting at node k, then $\vec{x} = [0, 0, ..., 1, 0, ..., 0] \in \mathbb{R}^N$, only $\vec{x}(k) = 1$. If the probability of ending in a certain node after n time steps need to be calculated, then:

$$P^n\vec{x}(t=0) = B^{-1}\Sigma^n B\vec{x} = \vec{x}(t=n) \tag{4.14}$$

Where Σ is a diagonal matrix and B is a change of base matrix that uses $P's$ autovectors. The diagonal values in Σ will be the autovalues of P. When $n \to +\infty$, one would still like to have a non zero probability of being in any state, otherwise that would mean that there is no feasible path between two nodes, and that would violate one of the conditions for the proposed network. If eigenvalues are lower than 1, their respective position in Σ^n will converge to 0. Eigenvalues greater than one are not possible because the resulting vector $\vec{x}(t=n)$ needs to add to 1. As a result, only an eigenvalue of 1 would survive as $n \to +\infty$. The eigenvector for that eigenvalue will be stationary state of the network. Only one eigenvalue can be 1, otherwise depending on the starting state $\vec{x}(t=0)$, convergence can be achieved to more than one different stationary states, and that is not allowed. A *Matlab* function was programmed to find a random connection matrix, where the matrix is made sure to be symmetric (each connection between two nodes goes both ways) and that each column or row has only M positions different than 0. Additionally, the existence of a unique eigenvalue equal

to 1 was required.

Using the before-mentioned constraints, the case of different networks with 32, 64, 128 and 256 nodes with completely random connectivity and M equal to 4, 6, 8 and 10 is presented. The idea with these simulations is to empirically show that any network satisfying the before-mentioned constraints will always work. For these network simulations, every node was made sure to inject a packet with random destination every time one of its inputs was empty, or the packet arriving to a node was locally delivered. Consequently, the simulated network will always have all of its links full, so the most possibly congested network situation is being considered here. In Table 4.1 and 4.2 the mean delay and mean throughput obtained for these networks is shown. Simulation was run for almost 100K time steps, and all of the injected packets were delivered without any dead-locks, live-locks or packet drop. Each value in Tables 4.1 and 4.2 are the mean of 256 simulations from different random networks with the same M and N. Additionally, one of the most used network topologies was simulated, a mesh network, where $M = 4$. Height and width for the mesh were limited to 4, 8, 16, 32, 64 or 128, and all possible combinations were simulated. Edge effects were considered here, making the corner nodes have connections to only two other nodes, and the side nodes connections to only 3 other nodes. This additionally shows empirically the validity of the previous demonstration that the number of connections for the nodes in the network can be M or less. For all the simulations performed, at some point the injection of packets into the network was stopped and it could be seen how

M \ Nodes	32	64	128	256
4	4.06	5.36	6.73	8.11
6	2.95	3.90	4.85	5.83
8	2.48	3.39	4.09	4.88
10	2.24	2.91	3.67	4.36

Table 4.1: Mean delay suffered for packets injected into a fully loaded network (all links are used all the time). Case of a random connecting network with different number of nodes. M is the number of connections each nodes connects to.

M \ Nodes	32	64	128	256
4	25.38	40.41	66.70	113.51
6	48.64	78.52	131.69	226.16
8	73.49	119.22	201.64	349.65
10	98.44	163.55	274.60	478.76

Table 4.2: Mean throughput for packets injected into a fully loaded network (all links are used all the time). Case of a random connecting network with different number of nodes. M is the number of connections each nodes connects to.

all the remaining packets get delivered until the network becomes completely empty.

In Tables 4.3 and 4.4 the results for the case of the mesh network are presented.

4.2.4 Self-Diagnosis in the *L2 network*

The read and write word size for the external DDR memory is 256 bits, and then it was desired to keep that same data width for the *L2 network*. PUs might receive data from the DDR memory, make a few changes or not to it, and forward to the *L2 network*. This process would become much more simpler if both *L1 and L2 network*

Horizontal \ Vertical	4	8	16	32	64	128
4	5.00	8.17	15.50	29.65	55.56	99.08
8	8.17	11.52	18.23	32.32	60.21	105.36
16	15.50	18.23	25.16	38.17	63.62	110.50
32	29.65	32.32	38.17	51.37	74.21	117.71
64	55.56	60.21	63.62	74.21	99.49	136.45
128	99.08	105.36	110.50	117.71	136.45	175.65

Table 4.3: Mean delay suffered for packets injected into a fully loaded network (all links are used all the time). This table shows the case of a mesh network with different number of nodes in the horizontal and vertical direction.

Horizontal \ Vertical	4	8	16	32	64	128
4	8.00	11.38	13.20	14.59	16.18	18.72
8	11.38	18.00	24.41	28.90	32.25	37.75
16	13.20	24.41	37.36	51.15	63.40	74.98
32	14.59	28.90	51.15	78.53	111.63	144.71
64	16.18	32.25	63.40	111.63	171.21	253.98
128	18.72	37.75	74.98	144.71	253.98	401.22

Table 4.4: Mean throughput for packets injected into a fully loaded network (all links are used all the time). Maximum throughput achieved in a mesh network of different number of horizontal and vertical nodes.

had matching words lengths. That was the idea behind the choice of the packet size.

When physically building this network, due to the high number of node connections, chances are that there might be one or more faulty connections from one node to another, and then a way of identifying these broken connections should be considered. Keeping in mind the requirements for the network mentioned before, if a link from node a to node b is broken, not only that link has to be removed from the routing tables, but also the link from b to a, so that the number of incoming and out-

going connections in a node stay the same. Following power up, all of the nodes will receive a global reset signal, and after this, a second global signal will command the nodes to self-diagnose their network links. After removing the faulty links from the routing tables in the nodes, one could end up with a fully connected network, where any node can reach any other node, or it might happen that some of the nodes might be completely isolated from others. One of the nodes in the network has a processor (more specifically *ARM M0 processor*) that will coordinate the activity among all of the processing units connected to each of the nodes. If this Fishbone, because of the existence of broken links, has no access to any processing unit in a particular node, then that processing unit has to be taken out of the pool of available processors. A diagram of a case in which this can happen is shown in Figure 4.10. With a red cross the broken links that make the set of nodes A and B be completely isolated from each other are shown. Because the connection from the *coordinating processor* to the nodes in A is only through B, processors in the set of nodes A become useless for the *coordinating processor*. In figuring out which processors are reachable, ping packets are sent to all of the nodes, and if a response is not received from a certain node, this means that the mentioned node is unreachable. This ping packet will have a counter, and after a certain amount of time, if it didn't reach it's destination, then this packet will be dropped, otherwise this packet would remain in the network forever, based on the mathematical development mentioned in 4.2.2. The counter used to determine when a ping is dropped is actually the *time counter*.

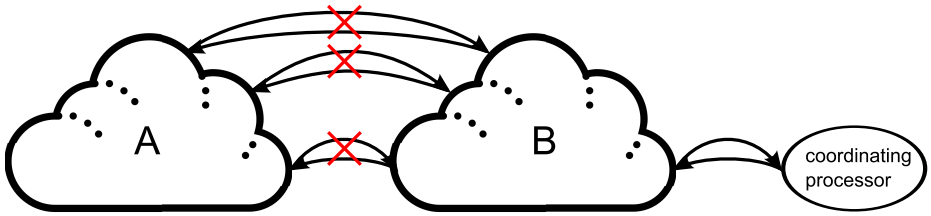


Figure 4.10: Isolation of PUs. The broken links show how processors in nodes set A are useless to the *coordinating processor*.

Now one needs to consider how is it that a link is concluded to be broken. A link from node a to node b is considered to be broken if for any of the lines in its bus, node b cannot read both logical states '0' and '1' for each of the lines. If a line is broken, it is constantly either set to '0', '1' or it might be floating, in which case it will still be read as a constant '1' or '0'. Because the bus considered will be higher than 256 lines, a very simple and compact architecture for performing this self-diagnose should be considered. A custom asynchronous cell was design for this purpose, and it is called *SEN_SENSE*. In Figure 4.11 the architecture communicating node a to node b is presented. The output registers from node a are registers that are reset using an asynchronous reset signal (global reset). These registers will have the capability of being configured as a shift register. A second global signal will indicate node a to inject a '1' into the shift register, and then a pulse of one clock cycle will be propagated from the left-most output register to the right-most register in Figure 4.11. The idea of the *SEN_SENSE* cell is to receive a pulse through its left input, and after receiving a second pulse through its right input, a pulse very similar to

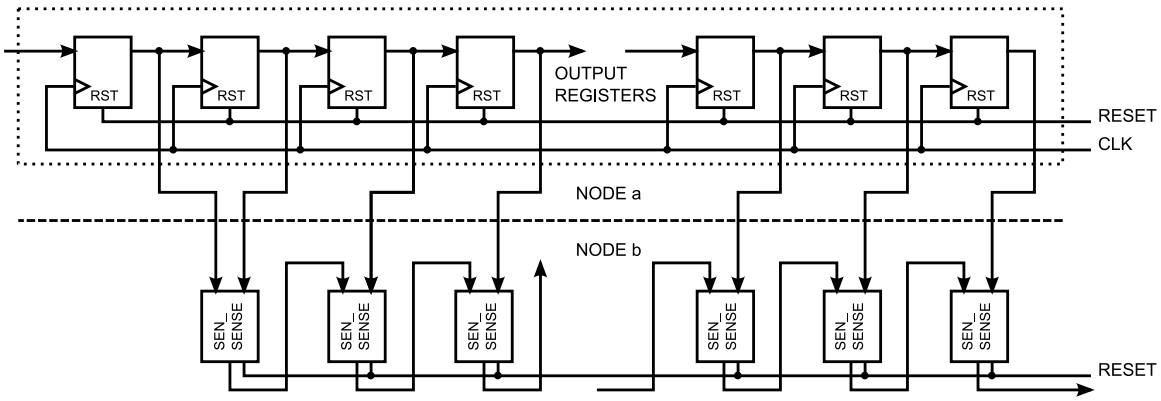


Figure 4.11: The node self-diagnosing mechanism. A pulse is sent through the top shift register belonging to node a. Node b will receive the first pulse through the left input of the most left *SEN_SENSE* block. Upon receiving that pulse, the second pulse received through its other input will be forwarded to the output of the cell into the first input of the following *SEN_SENSE* cell.

this second pulse, but delayed, will be propagated to the first input of the following *SEN_SENSE* block. If a pulse is received at the output of the last *SEN_SENSE* block, then the link is considered to be healthy, otherwise it is broken.

Figure 4.12 shows the architecture for the *SEN_SENSE* cell. A link can be broken, as mentioned before, by being permanently set to a logic '0' or '1'. At the top part of the circuit, two signals are generated to drive the RS flip-flop with outputs $Q1$ and $\bar{Q}1$. Upon receiving the $RESET$ signal, all of the output registers from the links between nodes are set to zero. If for any reason $X1$ link remains at logic '1' while $RESET$ is high, this means that the link has to be broken. It is for this reason that, if this happens, $Q1$ = '0', not allowing signal $aux1$ to propagate to the final gate, preventing any pulse from $X2$ to be forwarded. If $X1$ happens to be at the logical state '0' when $RESET$ = '1', a transition from '0' to '1' needs to still be

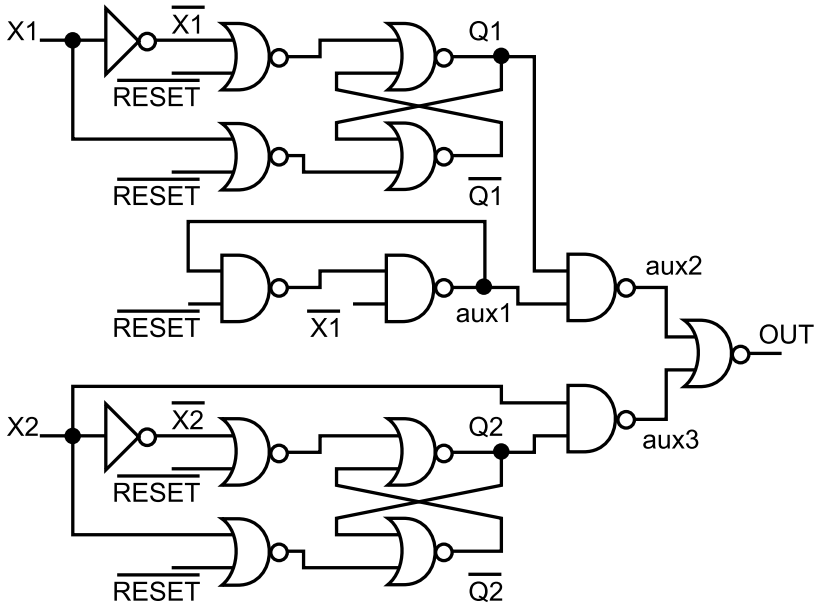


Figure 4.12: The *SEN_SENSE* cell architecture. The asynchronous circuit involved in the network self-diagnosis is presented.

sensed in order to allow the pulse coming from $X2$ to be forwarded. If $X1$ = '0' while $RESET$ = '1', $aux1$ = '0'. If $RESET$ goes low, that value will still be maintained. Only if $X1$ transitions from '0' to '1' it is that $aux1$ = '1' and the signal coming from the other $X2$ branch can be forwarded to the output. In the $X2$ branch exactly the same idea is applied for the RS flip flop, but if $Q2$ = '1', then $X2$ is forwarded to $aux3$, and if no problem was found in $X1$, then $aux3$ = '0' and $X2$ is forwarded to OUT. As it can be seen from Figure 4.11, if any of the *SEN_SENSE* blocks does not forward a pulse, then the final output of the whole chain will not output a pulse, and it can be concluded one is in the presence of a broken link.

4.2.5 *L2 network* Routing Tables

As stated before, the only requirement for the routing of packets in the network, is that routing tables should be defined in such a way that a feasible path from any PU to any PU can be always found. Routing tables could become very complicated, a routing table input could actually be defined for every single node in the network. This kind of approach would be terrible if the network node is desired to be as compact as possible. Considering that a mesh network was chosen for the CMPs, then, by assigning the x and y coordinates as the network node addresses, then the routing task can be simplified. Let's take as an example Figure 4.13. Assuming a packet is allocated in node $(4, 2)$, four possible destinations are colored in green, blue, purple and yellow. Depending on where the packet needs to go, two simple options are available. For example, if the destination is north-west, then north or west interfaces are feasible options. As a first approximation, this scenario would only require the use of two comparators for each input interface to a node in order to define were that packet needs to go.

In the network, with each clock cycle, a packet travels from one node to a neighboring one. This kind of constraint leaves no room for pipelining, and if $300MHz$ is the desired running frequency for the network, then the logic used in routing packets needs to be very efficient. On top of this, the traveling packets from one node to the neighboring one, need to travel long distances, making the addition of high capacitance lines be another problem that needs to be considered. This is depicted

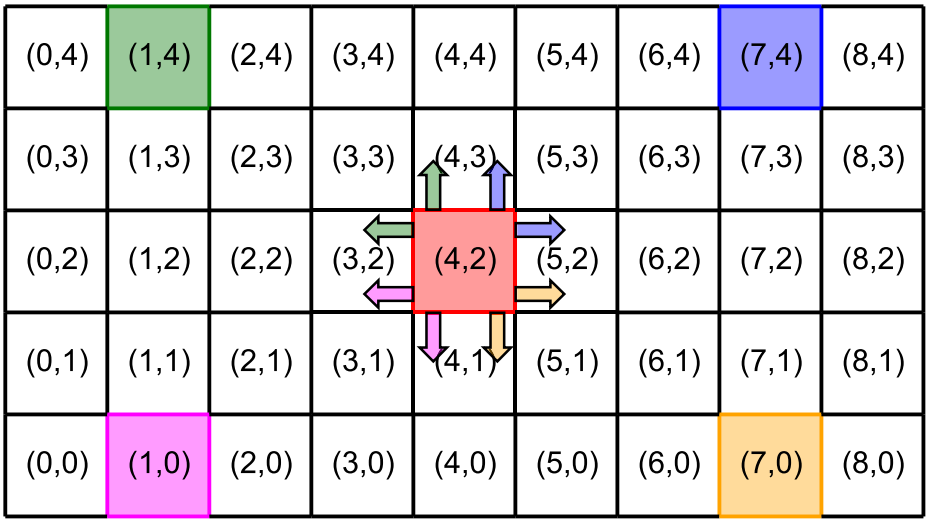


Figure 4.13: Network address topology. The network node addresses are mapped according to their position in the network. With a packet in network node $(4, 2)$, and four destinations shown in green, blue, purple and yellow, the output interface to where the packet should be sent in node $(4, 2)$ can be easily determined by comparing the destination address with the local address. Two possible options are found for each destination, unless the destination node shares the same column or row.

in Figure 4.14, where it can be seen that the logic determining where a packet is delivered is completely combinatorial. The combinatorial cloud from the center node receives five inputs (four from the neighboring nodes and one coming from the local PU) and generates five outputs (four outputs going to the neighboring nodes and one assigned to the local PU).

When looking at a mesh network, different types of nodes can be found, the internal ones and the ones on the boundary. On top of this, the ones on the boundary can be also divided into the ones that are vertices or not. Depending on what type of node is analyzed, 2, 3 or 4 interfaces can be defined. This suggests that three different types of network nodes will have to be designed. In order to make the design easier, a

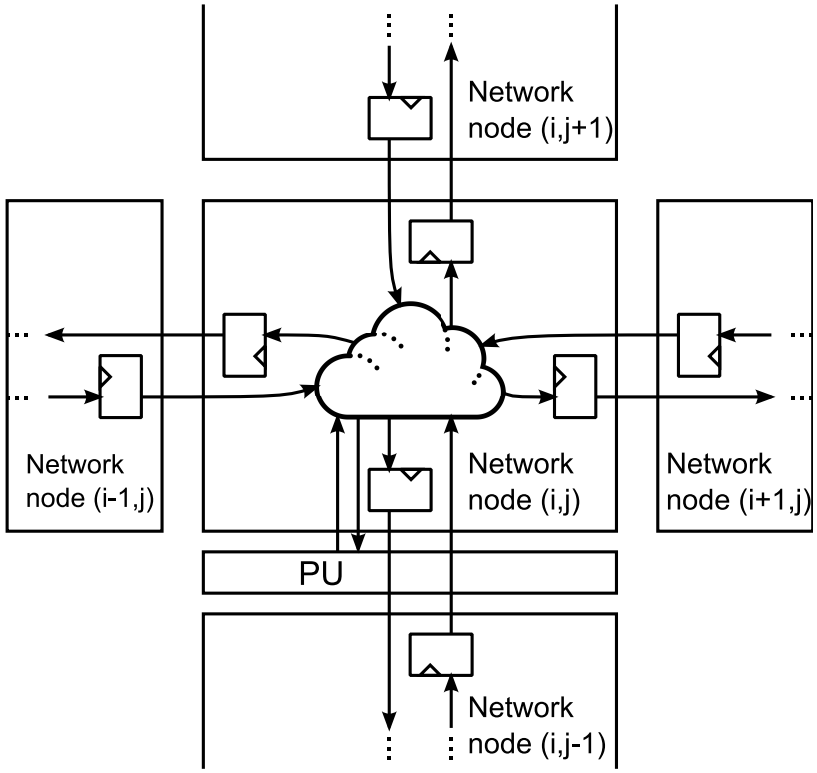


Figure 4.14: Combinatorial routing. This figure shows how the routing of packets in the network does not leave any room for pipelining. With every clock cycle a packet, unless locally delivered, hops one node.

single network node was decided upon. This network node has local connection to its PU, and four other connections to neighbors. The connections to the neighbors will be defined as active with the usage of four control signals. These four input signals will be called $N\_link\_down\_i$, $W\_link\_down\_i$, $E\_link\_down\_i$ and $S\_link\_down\_i$. The internal nodes will have these signals hardwired to '0'. Depending on where in the boundary a node is, these signals will be hardwired differently. For instance, the top-left vertice will disable the north and west interface by setting $N\_link\_down\_i$ and $W\_link\_down\_i$ to '1'. In the previous section the self-diagnosing mechanism was explained. If a link was damaged, then the two links connecting the involved nodes

will be disabled. There will be internal signals determining if a link is healthy or not, so it will be the OR logical operation of these signals and the corresponding input control signals *N_link_down_i*, *W_link_down_i*, *E_link_down_i* and *S_link_down_i* that will decide the usage of the different interfaces.

In Figure 4.15, a very general idea of how the *Network 2* node is organized is presented. All four neighboring interfaces, plus the local PU interface, have a *Packet Desired Destination* block that calculates the interface through which the incoming packet should be routed. This block takes into account the state of the links to the neighboring nodes through the $*$*_link_down_i* inputs, but it does not take into account any other possible incoming packets, meaning that none of the *time counters* or *fractional counters* are being analyzed at that point. If one looks at the output signals from these blocks, with the exception of the one belonging to the local PU, it can be seen that five signals are generated, *send_X_N*, *send_X_W*, *send_X_E*, *send_X_S* and *send_X_L*. For all of these signals X can be N, W, E, S or L. All of these signals encode in a one-hot encoding the interface to which a packet desires to be routed. As an example, if *send_W_N* = '1', this means that the packet coming from the west interface should be routed to the north interface. For the case of the local PU block, only four outputs are generated because the packet coming from a PU does not go back to the same PU. The way routing is performed in these blocks is different from the first order approximation mentioned before. Routing is based on the interface where the packet comes from, meaning that all of the *Packet Desired*

Destination blocks will be different according to the interface to which they belong. The routing mechanism is presented in Figure 4.16. In this Figure the north interface routing is shown, and the cases of all the other three neighboring interfaces can be obtained by just rotating the depicted nodes in Figure 4.16. In tables from Figure 4.16, the existence of some ambiguity in routing can be seen for some cases. In those cases, random hardwired numbers local to each network node are used to choose among the different options. The block *Local Delivery Priority* analyzes, according to the priority of the incoming packets, which of them can be delivered to the local PU. The ones not delivered will be sent back to the network. Again, four output signals in one-hot encoding are generated to indicate which packet should be delivered locally, $deliver\_loc\_N$, $deliver\_loc\_W$, $deliver\_loc\_E$, $deliver\_loc\_S$, and an additional signal $deliver\_loc$ that indicates the existence of a packet. These signals will be used in a multiplexer to select which of the packets needs to be forwarded to the local PU. The output of the multiplexer has been labeled in Figure 4.15 as *packet to local*.

As seen before, one-hot encoding is being used for many of the blocks, and the reason behind it, is that the logic inferred to map the encoding process suffers less propagation delay, at expense of using more area. Because all of the logic involved in routing cannot be pipelined (in fact, it is completely combinatorial, no registers are present), and a frequency of at least $300MHz$ is desired for the network, truth-table guided synthesis had to be avoided. Every single operation performed for routing had to be carefully crafted. Scripts in *Matlab* were necessary to write the *VHDL* code for

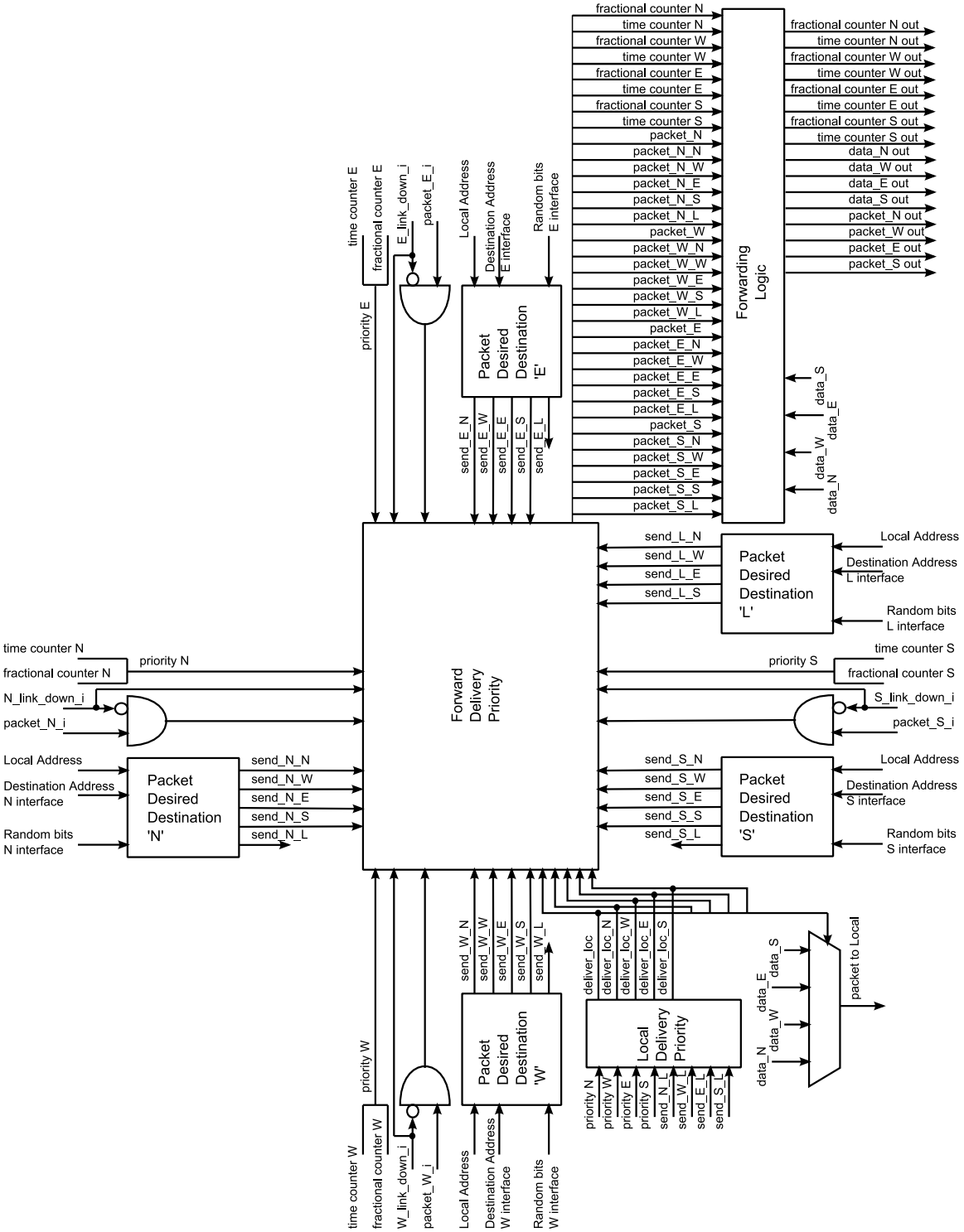
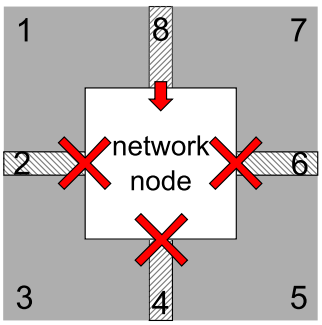
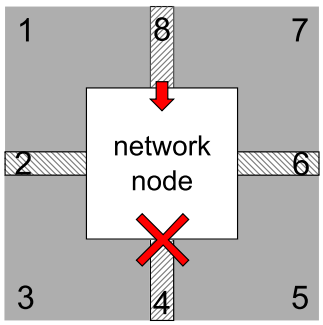


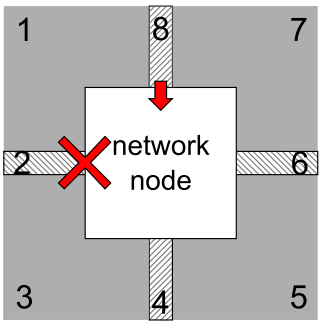
Figure 4.15: *L2 network* node. *L2 network* network node diagram.



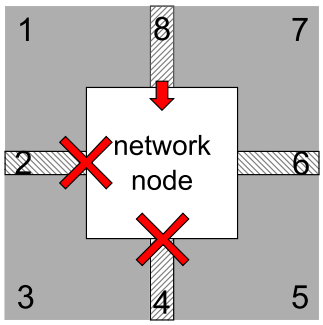
Target direction	Routing Interface
area 1	W
area 2	W
area 3	W
area 4	S
area 5	E
area 6	E
area 7	E
area 8	E or W



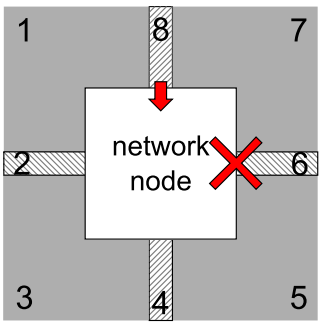
Target direction	Routing Interface
area 1	W
area 2	W
area 3	W
area 4	E or W
area 5	E
area 6	E
area 7	E
area 8	E or W



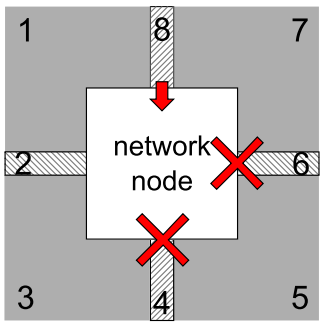
Target direction	Routing Interface
area 1	S
area 2	S
area 3	S
area 4	S
area 5	E
area 6	E
area 7	E
area 8	E



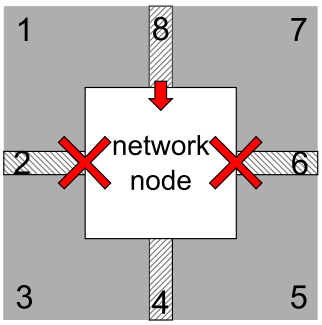
Target direction	Routing Interface
area 1	E
area 2	E
area 3	E
area 4	E
area 5	E
area 6	E
area 7	E
area 8	E



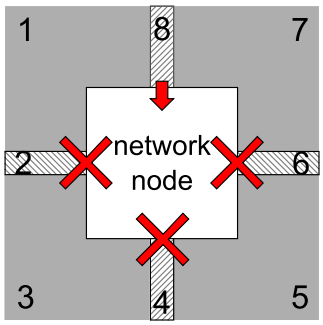
Target direction	Routing Interface
area 1	W
area 2	W
area 3	W
area 4	S
area 5	S
area 6	S
area 7	S
area 8	W



Target direction	Routing Interface
area 1	W
area 2	W
area 3	W
area 4	W
area 5	W
area 6	W
area 7	W
area 8	W



Target direction	Routing Interface
area 1	S
area 2	S
area 3	S
area 4	S
area 5	S
area 6	S
area 7	S
area 8	S



Target direction	Routing Interface
area 1	N
area 2	N
area 3	N
area 4	N
area 5	N
area 6	N
area 7	N
area 8	N

Figure 4.16: *L2 Network* routing tables. Routing tables for the case of the north interface. Different cases for when links are down are considered. The routing tables for all the other interfaces can be extracted by just rotating the node figures matching the output routing interfaces correctly.

the routing logic. The resulting *VHDL* file describing the behavior of the middle block *Forward Delivery Priority* in Figure 4.15 turned out to be close to 25000 lines of code, and needed to be synthesized with the *exact map* option. Considering the priority values of all of the incoming packets to a node, and the interfaces to where these packets want to be forwarded, this block applies the priority based routing to identify where packets need to go. If a packet needs to be routed incorrectly, the packet will be forwarded to the closest interface to the originally desired one whenever possible. This middle block will identify where packets need to be forwarded by using again one-hot encoding outputs (the ones called *packet_*$*$) and will additionally update the *time counter* and *fractional counter* of these packets. This block will discard packets that are being delivered locally from being forwarded to the network. Finally the *Forwarding logic* block will take all of the data packets and by using the outputs from the *Forward Delivery Priority* block, these packets and counters will be forwarded to the correct interfaces.

The delivery of packets to the local PU is done through a four-phase handshaking protocol, meaning that an acknowledge signal is expected back from the PU to confirm that the packet sent was actually received. This makes the delivery of a packet take more than a clock cycle, and then if while waiting for the acknowledge from the PU, an additional packet arrives to the node and wants to be delivered, this packet will be sent back into the network until the on-going local transaction has finished. Furthermore, a similar situation arises when several packets desire to be locally delivered and only

one can start the transaction to the local PU. All of the remaining packets will be forwarded back into the network. This makes the delivery time of a packet increase. Additionally, demonstrations shown in the previous sections do not consider bouncing of packets into the network. Fortunately, this bouncing effect does not affect the characteristics of the network or the priority routing explained before. Any packet sent back into the network will keep its priority with respect to all of the packets present in the network. If one looks at the rejected packet that was injected back into the network, the fact that the packet was rejected is not very important, because the packet has a certain destination and priority, and with that, one has enough to route it back to its destination. Because the rejection of packets is very much linked to the kind of processing and traffic in the network, the only consideration that had to be taken was to increase the number of bits used for the *time counter*. Considering that a packet could bounce several thousands of times (very unlikely), the *time counter* was set to be 24 bits. With respect to the *fractional counter*, the number of bits required does not change and it is $ceiling(logN_M) = 4$ (the number of nodes in the network will be $N = 128$ and the number of interfaces each node has is at most $M = 4$).

Unless all of the links but one are down in a node, a packet arriving to that node will never be forwarded back to the interface it came from (unless priority dictates it). This routing constraint is actually very useful because it allows the communication to PUs that have been almost isolated due to broken links in the network. It actually allows to find PUs by using the method of solving labyrinths by always "following a

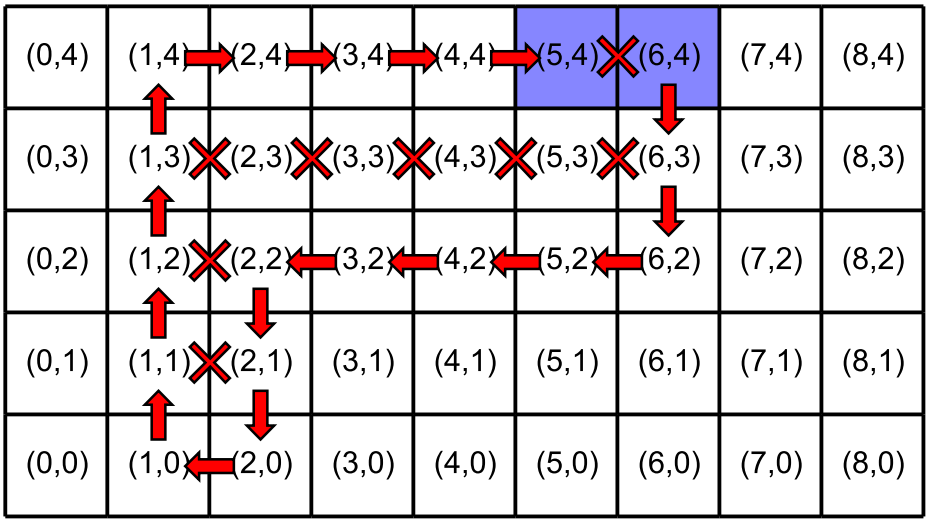


Figure 4.17: Routing with broken links. Packet in node $(6, 4)$ has to be delivered to node $(5, 4)$. By following the routing rules from Figure 4.16, the packet finds its way to the destination node by mimicking the labyrinth solving strategy of always "following a wall". Broken links are shown with an X.

wall". An example is given in Figure 4.17 where, if routing based on Figure 4.16 is used, the packet will be routed in a loop until it reaches its destination.

A final comment about this *L2 network* node is that, even if two network sizes will be implemented for the four CMPs, the network node for the *L2 network* will be the same for both 64 and 128 PUs networks. The reason for this is that the 64 PUs network has 8 instead of 16 PU along each row, and then this network can be considered a reduction from the bigger 16x8 PUs network. The number of bits used for vertical and horizontal addresses in the case of the 16x8 PUs network is compatible with the number of bits required in the smaller network. The 64 PUs network will just look like a trimmed version of the 128 PUs one.

4.3 The *L1 & L2 network* Node

4.3.1 Overall *L1 & L2 network* Node Description

In sections 4.1 and 4.2 a description of the functioning of the two networks on chip was presented. Following the strategies presented in Chapter 2, two network nodes containing both the *L1* and *L2 network* nodes were synthesized. The original idea was to have both cases presented in Figure 4.3 in a single node, and choosing one of the two configurations by just hard-wiring an input bit. Unfortunately, that approach tampered with the achievable clock speed, and then it was decided to synthesize two network nodes with both cases. For these two network node designs, the *L2 network* node is exactly the same, the only difference is the one mentioned for the *L1 network* node.

Figure 4.18 shows the layout for both network nodes. The size of the network node and its PU is $900\mu m$ by $1580.4\mu m$, where the network node takes only around 18% of the area. Across all of the CMPs, several power domains were used with the objective of reducing the power consumption as much as possible. For instance, on the network side of the CMPs (not the *DDR DRAM PHY* side), two main power domains have been used. One of them is the one supplying power to both networks on chip (called *VDD_NET*), and the second one, is the one available to all of the processing units which provides a lower voltage (called *VDD_PU*). The availability of an additional power supply *VDD_PU* necessitated the usage of level shifters to convert signals on

the *VDD_NET* to the *VDD_PU* domain and vice versa. For these level-shifters a few options were considered,[23–25] but unfortunately many of the approaches needed very specific transistor sizing or relied on the usage of biases. The architecture presented in[26] was used, as it did not present any static power dissipation (bias-less) and device sizing was not required for its basic operation. With this level-shifter, conversion from any voltage to any voltage (in both cases $\leq 1.2V$) could be performed without the need of different level-shifter versions. The only requirement for this design is that a differential input is needed for the shifted signal.

The NoCs were designed to run as fast as possible, and because of the combinatorial constraints shown in Figure 4.14, a voltage lower than the nominal one (1.2V) was not an option. On the other hand, some of the processing units might not need to go as fast (some of them could go as slow as 1MHz), and then, since power is linearly dependent to the square of the voltage, lowering the power supply as much as possible for those PUs that could handle it, was very tempting. Unfortunately, some processing units might actually need to go fast, needing the 1.2V supply, and then a solution had to be thought about this problem. The solution found was actually providing the signals going from the network node to the PU in both power supply voltages (the ones in the *VDD_PU* power domain requires level-shifters). With respect to the signals going in the opposite direction, even in the case in which it is not required, level-shifters were placed so that no matter what voltage domain the PU used, the level-shifters will always provide the correct voltage levels to the network

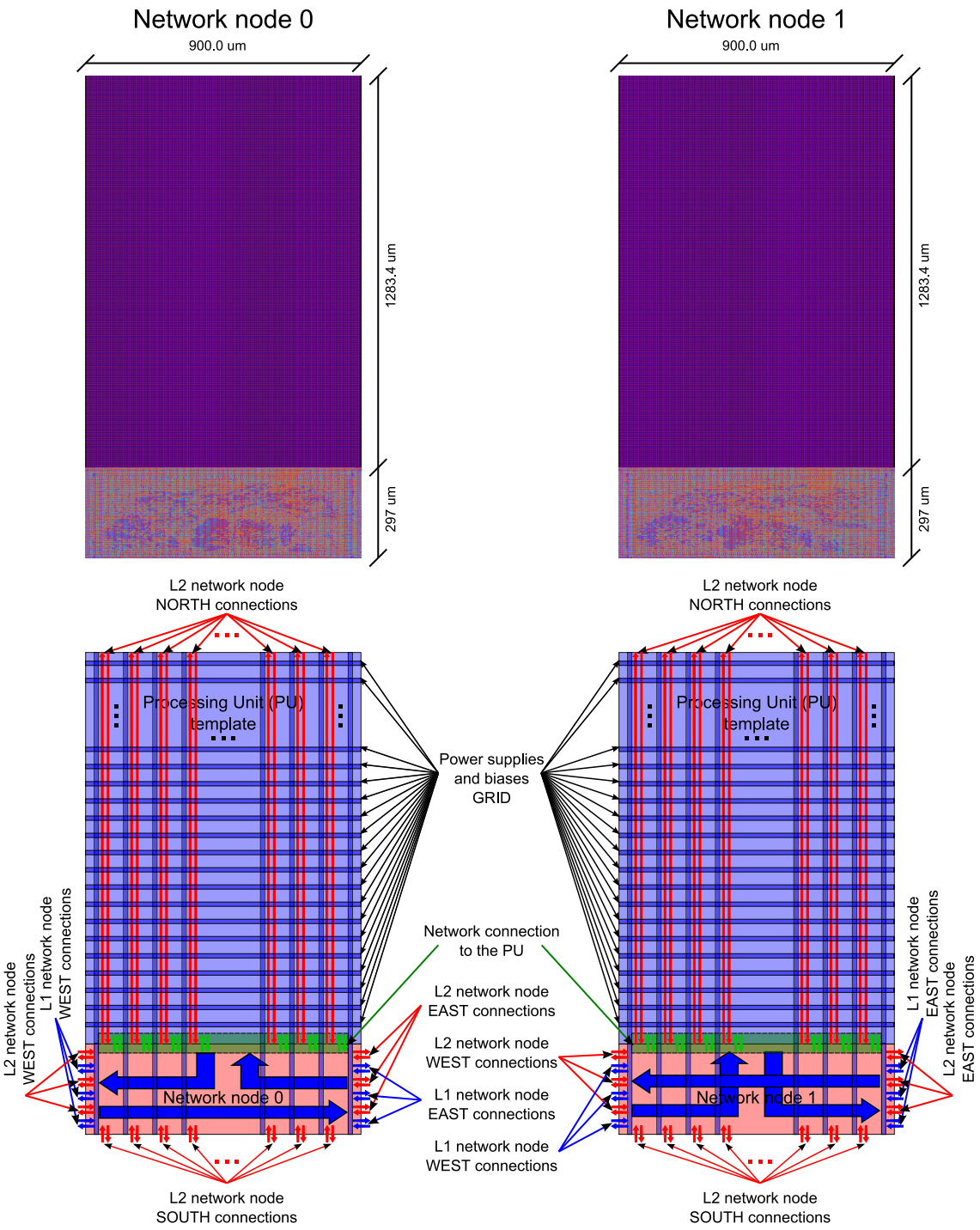


Figure 4.18: Network node layout. This figure presents the layout for the two types of network nodes used in the CMPs. Each node contains both *L1* and *L2* network nodes.

node. These level-shifters are incorporated as part of the network node so that the PU design could be simpler.

In addition to the voltage supplies, several external and internal biases were generated for the PUs that required them. All of these signals would be distributed, as mentioned before, across the chips using very low resistance power grids, where a piece of it would be available to the PU. Because of all of these constraints that the PUs needed to handle, a template with the position of the power supplies and biases, and the position of the pins connecting the network node to the PU was designed. This template would be recreated with a *tcl* script, and it would be the starting point for anyone designing a PU. The person designing a PU would have the grid positions for all of the power supplies and biases, along with the positions of all of the signals communicating the PU with its network node. The choice of the power domain could be taken by the designer and no change on the network node side needs to take place. The PU designer would have the power distribution and pin assignment completely solved, which is very convenient, because generally the design of the power distribution and pin assignment is not a trivial matter.

The metal stack used for these CMPs is composed of a set of three different metals. The first six metals (M1 to M6) have almost the same characteristics, and they are usually used for the routing of high speed signals. The next two metals (EA and EB) are wider ones, and have more capacitance than the previous six ones. Finally, the last metal (LB) is the thickest one and it is the one generally used for the

bondpads and C4 bumps. On the bottom two diagrams in Figure 4.18, the vertical and horizontal blue lines represent the different power supply voltages, ground and the biases available to the PU. These signals were designed in metals EA (vertical) and EB (horizontal), leaving M1 to M6 available for the internal routing of the PU. It can be seen that along with the grid, the signals belonging to the *L2 network* node NORTH connections are shown. These signals had to share metal EA with the power distribution grid to connect a node to the one on the top. The practice of using EA as a metal for routing high speed signals is not the most recommended one because of the high capacitance of these metals, but it was the only way of warrantying the same number of horizontal and vertical metals available for the PU. The transmission delay suffered by signals in this metal EA, running for more than a millimeter without buffering was the main challenge when trying to reach high speeds for the NoCs. In red, on Figure 4.18 the connections to all of the neighboring nodes are shown for the *L2 network*. In blue the connection to the *L1 network* node to the right and left are shown. The big blue arrows represent the difference between the two network nodes presented in this figure as seen in 4.3. In green the connection between the PU and its network node is shown. It is on that green island that the mentioned level-shifters are placed.

As described before, the network data packet size for both networks on chip was set to 256 bits, considering every connection bidirectional, over 4000 pins had to be distributed on the perimeter of the node, making the *Place & Route* task a very

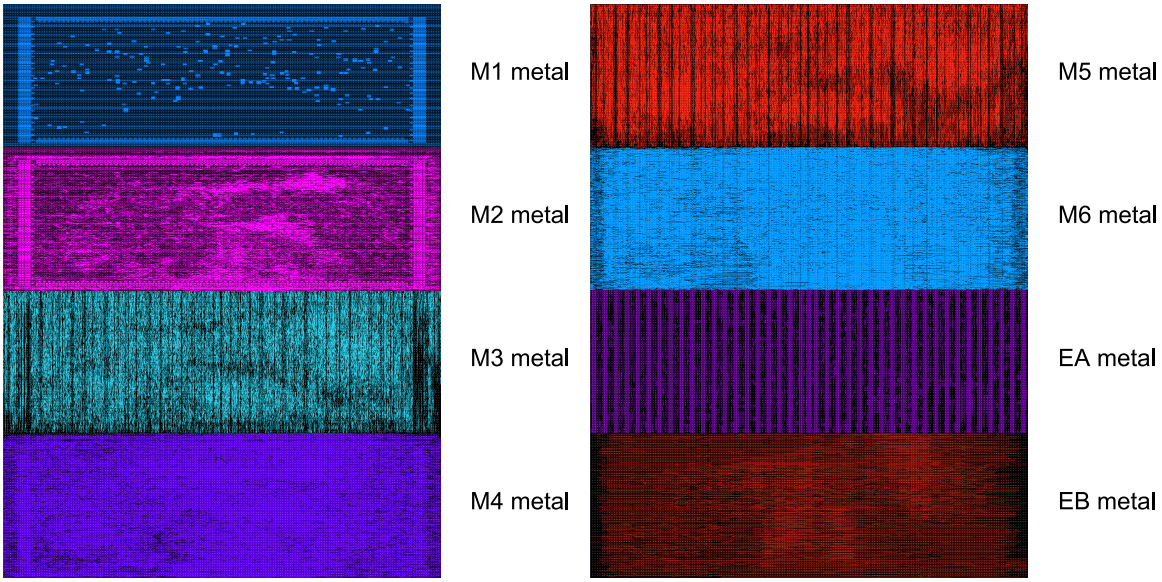


Figure 4.19: Routing density achieved on the network node. Over 90% of the metals' routing capability was reached for all of the metals used in routing inside of the network node.

difficult one, if one wants the used area to be as small as possible and the clock speed as high as possible. Because of the high congestion of wires in the network node, manual placement of many of the cells had to take place. Some of these cells are the *SEN_SENSE* diagnostic cells and cells driving the outputs of the node. Placement of specific driver sizes were needed for these outputs because of the needed equalization of input/output delays in the node as seen in Figure 2.1. Figure 4.19 shows the wire congestion found for each of the metals used for routing in the network node. The regularity found for metals M1 and M2 is due to the manual placement of cells mentioned. All of the shown metals reached over 90% of their routing capability and there was almost no need to insert metal fillers, which is usually very hard to accomplish.

Because of the high importance of starting both NoCs at a known state, all of the registers used for the network node design feature an asynchronous reset. This is the reason why a reset tree was mentioned in 3.2. A synchronous reset could have been an option if designed properly. The problem for this synchronic reset is that the reset signal would have to be pipelined all the way up to every network node with the same delay, which turns out to be not a simple task at all when modular designs such as this one are considered.

Figure 4.20 shows a small example of how the shown network node can be tiled to generate both NoCs. In blue the *L1 network* node connections, and in red the *L2 network* node connections. On the right and left ends, the *L1 network* node connections are fed back into the network to generate the ring networks. The address assigned to each network node is hardwired on the top-level design of the chip as shown through the inputs *hor_addr_i* and *ver_addr_i* in each of the nodes. The routing tables of the *L2 network* are adjusted according to the input signals *healthy_N_i*, *healthy_S_i*, *healthy_W_i* and *healthy_E_i*. A one in any of those input would disable the corresponding interface. The nodes on the top row have *healthy_N_i = '1'*, the ones on the bottom row *healthy_S_i = '1'*, the ones of the right column *healthy_E_i = '1'*, and the ones of the left column *healthy_W_i = '1'*.

It was expressed before that the clock received by the PU was a programmable one, and then the network clock and the PU clock would not be in phase, necessitating the usage of a communicating protocol between the network node and the PU that

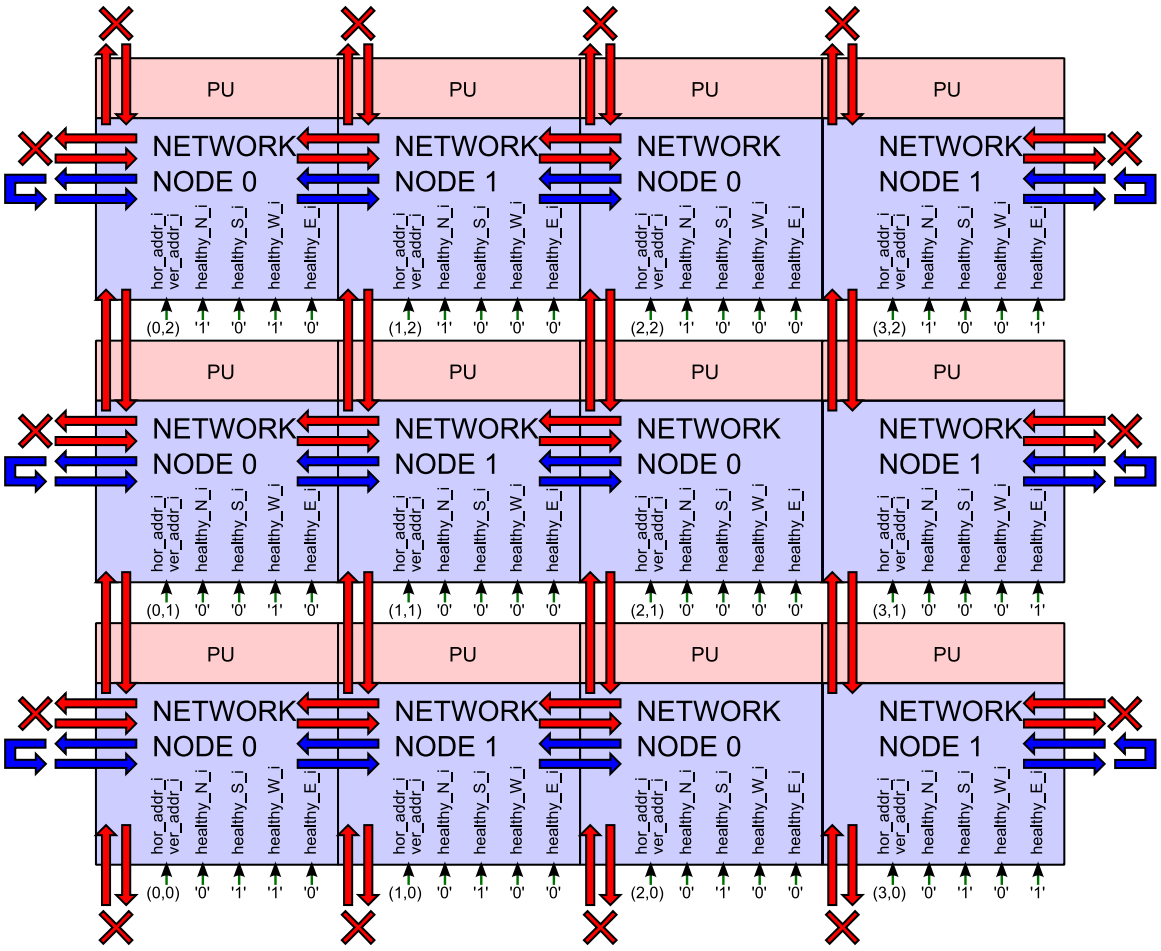


Figure 4.20: Example of usage of the network node. Example of a 4x3 network. The two types of network nodes have been alternated on each row.

doesn't rely on any particular clock. Even in the case of programming the PU clock to be the same one as the network clock, same phase cannot be assured because one has no control over the PU clock tree delay. An alternative could have been to constrain the clock tree delay internal to a PU to a particular value, but this approach would have only worked for the case the PU uses the *VDD_NET* domain, because a change in the power supply will change the propagation delay of the cells in the clock tree. Additionally, satisfying a clock tree delay constraint is very difficult to accomplish. For these reasons an asynchronous four-phase handshaking protocol was adopted. This approach would not be as fast as the case in which the network and PU clocks are in phase, but it is flexible enough to handle any clock frequency for either the network or the PU. Figure 4.21 shows the schematic and timing diagram for the four-phase handshaking protocol used in the unidirectional communication between a node and its PU. This protocol is used for a unidirectional communication, so in the case of the network node and its PU, two of these schemes will have to be used. When crossing clock domains, meta-stability could be a problem since the change of data on one side could coincide with the change of the clock used to latch that data on the other side. That could generate erratic behaviors explained in,[27] where the usage of a chain of registers would mitigate the problems. Usually a couple of registers would suffice, but in our case a chain of three registers was used for an increase of reliability. A *REQUEST* is elevated on the *BLOCK 1* side, and this *REQUEST* is converted into the *CLK2* domain through the usage of the chain of registers clocked by *CLK2*.

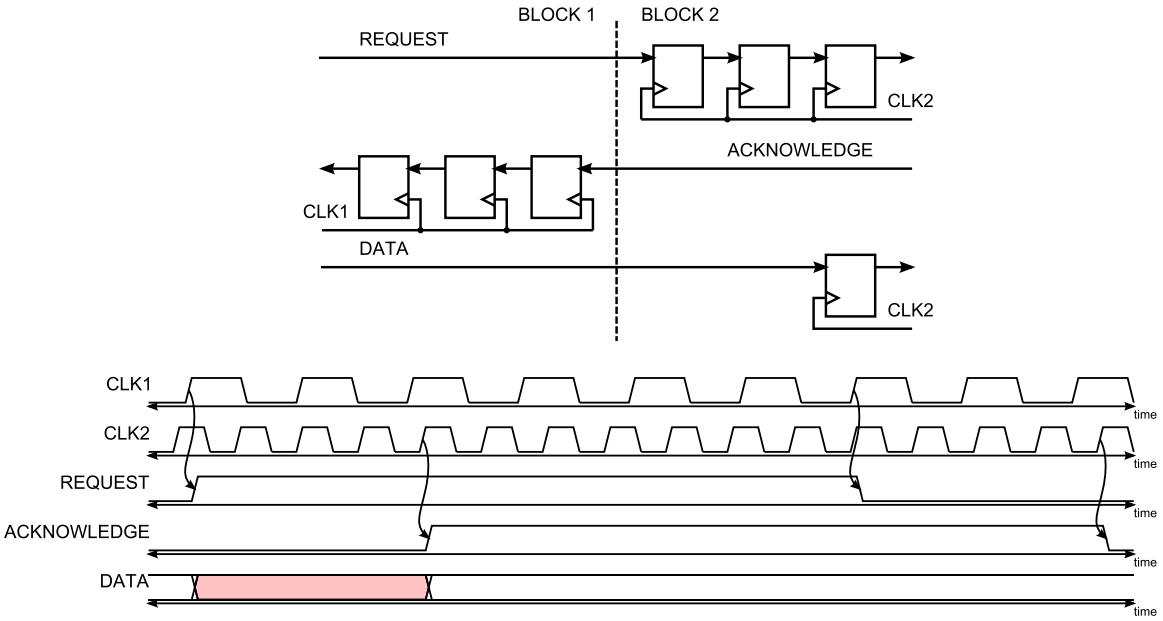


Figure 4.21: Four-phase handshaking protocol. Schematic and timing diagram for the asynchronous four-phase handshaking protocol. A request is elevated from *block 1*, and an acknowledge is expected back from the *block 2*.

Once the *REQUEST* is received by *BLOCK 2*, if this block is ready to acknowledge, an assertion in the *ACKNOWLEDGE* signal follows. This *ACKNOWLEDGE* signal is now converted to the *CLK1* domain through the usage of an additional chain of registers clocked by *CLK1*. Upon the reception of the *ACKNOWLEDGE* by *BLOCK 1*, this block proceeds to deassert the *REQUEST* signal, and when that deassertion is sensed by *BLOCK 2*, the *ACKNOWLEDGE* signal is deasserted as well. On this four-phase protocol data is sent from *BLOCK 1* to *BLOCK 2* without the usage of synchronizing registers. It is assumed that the *DATA* signal will not change its value until the *ACKNOWLEDGE* assertion is sensed on the *BLOCK 1*. This allows to play with the output/input delays when *Place & Route* takes place, allowing the signal *DATA* to change on the receiver's end at any time depicted in red in Figure 4.21.

4.3.2 Input/Output Signals

Table 4.5 shows the input and output signals interfacing the network node with the neighboring network nodes and the local PU for both *L1 network* and *L2 network*. Two different power domains were used for the network node, the one using VDD_PU ($< 1.2V$) and VDD_NET ($1.2V$). For outputs, the power domain is specified in parenthesis in table 4.5. For inputs the admitted voltage value is also specified in parenthesis.

Table 4.5: Description of the network node interface signals. Input and output signals found on each of the two network node versions in Figure 4.18. In parenthesis and in bold the power domain corresponding to the signal is shown.

Signal name	Bits	O/I	Description
clk_i	1	I	Network node's 300 MHz clock input. (**VDD_NET**)
reset_i & reset_n_i	1	I	Differential input. Asynchronous global reset for both NoCs. (**VDD_NET**)
diagnose_i	1	I	This signal controls the network nodes' self-diagnosis seen in 4.2.4. Upon power-up, a reset pulse is sent to all of the networks on chip, and after deasserting the reset, this signal is also pulsed, allowing the network nodes' self-diagnosis to begin. (**VDD_NET**)
Signals communicating the network node to the PU.			
PU_clk_VDD_PU_o	1	O	Programmable clock sent to the local PU. (**VDD_PU**)
PU_clk_VDD_NET_o	1	O	Same as PU_clk_VDD_PU_o. (**VDD_NET**)
PU_reset_VDD_PU_o	1	O	Reset signal for the local PU. A reset pulse of programmable width is received through a packet. (**VDD_PU**)
PU_reset_VDD_NET_o	1	O	Same as PU_reset_VDD_PU_o. (**VDD_NET**)
PU_enable_VDD_PU_o	1	O	Signal that tells the PU if the link between the local PU and its network node is enabled or not. (**VDD_PU**)
PU_enable_VDD_NET_o	1	O	Same as PU_enable_VDD_PU_o. (**VDD_NET**)
N2_hor_addr_VDD_PU_o	4	O	*L2 network* horizontal address. (**VDD_PU**)
N2_hor_addr_VDD_NET_o	4	O	Same as N2_hor_addr_VDD_PU_o. (**VDD_NET**)
N2_ver_addr_VDD_PU_o	3	O	*L2 network* vertical address. (**VDD_PU**)
N2_ver_addr_VDD_NET_o	3	O	Same as N2_ver_addr_VDD_PU_o. (**VDD_NET**)
NET_reset_VDD_PU_o	1	O	Input signal reset_i is forwarded to this output. (**VDD_PU**)

NET_reset_VDD_NET_o	1	O	Same as NET_reset_VDD_PU_o. (**VDD_NET**)
Path from the PU to the *L1 network* node.			
PU_N1_req_i & PU_N1_req_n_i	1	I	Differential input. Four-phase handshaking request signal from the local PU to its *L1 network* node. (**VDD_NET and VDD_PU**)
PU_N1_ack_VDD_PU_o	1	O	Four-phase handshaking acknowledge signal from the *L1 network* node to its PU. (**VDD_PU**)
PU_N1_ack_VDD_NET_o	1	O	Same as PU_N1_ack_VDD_PU_o. (**VDD_NET**)
PU_N1_tag_addr_i & PU_N1_tag_addr_n_i	23	I	Differential input. Tag address formed by combining N1_tag_addr_LR_i and N1_tag_LR_i. 23 instead of 27 bits because four bits are required to address a node in a token-ring network. (**VDD_NET and VDD_PU**)
PU_N1_op_i & PU_N1_op_n_i	1	I	Differential input. Desired operation to be performed by the DDR. 00 NOP, 01 READ, 10 READ ANSWER, 11 WRITE (**VDD_NET and VDD_PU**)
PU_N1_addr_i & PU_N1_addr_n_i	40	I	DDR address to which it is desired to write or from which it is desired to read. (**VDD_NET and VDD_PU**)
PU_N1_data_i & PU_N1_data_n_i	256	I	Data carried by the *L1 network* packet. (**VDD_NET and VDD_PU**)
Path from the *L1 network* node to the PU.			
N1_PU_req_VDD_PU_o	1	O	Four-phase handshaking request signal from the *L1 network* node to its PU. (**VDD_PU**)
N1_PU_req_VDD_NET_o	1	O	Same as N1_PU_req_VDD_PU_o. (**VDD_NET**)
N1_PU_ack_i & N1_PU_ack_n_i	1	I	Differential input. Four-phase handshaking acknowledge signal from the PU to its *L1 network* node. (**VDD_NET and VDD_PU**)
N1_PU_tag_addr_VDD_PU_o	23	O	Tag address formed by combining N1_tag_addr_LR_i and N1_tag_LR_i. 23 instead of 27 bits because four bits are required to address a node in a token-ring network. (**VDD_PU**)
N1_PU_tag_addr_VDD_NET_o	23	O	Same as N1_PU_tag_addr_VDD_PU_o. (**VDD_NET**)
N1_PU_addr_VDD_PU_o	40	O	DDR address to which it is desired to write or from which it is desired to read. (**VDD_PU**)
N1_PU_addr_VDD_NET_o	40	O	Same as N1_PU_addr_VDD_PU_o. (**VDD_NET**)
N1_PU_data_VDD_PU_o	256	O	Data carried by the *L1 network* packet. (**VDD_PU**)
N1_PU_data_VDD_NET_o	256	O	Same as N1_PU_data_VDD_PU_o. (**VDD_NET**)
Path from the PU to the *L2 network* node.			
PU_N2_req_i & PU_N2_req_n_i	1	I	Differential input. Four-phase handshaking request signal from the PU to its *L2 network* node. (**VDD_NET and VDD_PU**)
PU_N2_ack_VDD_PU_o	1	O	Four-phase handshaking acknowledge signal from the *L2 network* node to its PU. (**VDD_PU**)
PU_N2_ack_VDD_NET_o	1	O	Same as PU_N2_ack_VDD_PU_o. (**VDD_NET**)

PU_N2_is_data_i & PU_N2_is_data_n_i	1	I	Differential input. Indicator that the data content of the packet is data or a command. (**VDD_NET and VDD_PU**)
PU_N2_dest_hor_addr_i & PU_N2_dest_hor_addr_n_i	4	I	Differential input. Horizontal address in the packet destination. (**VDD_NET and VDD_PU**)
PU_N2_dest_ver_addr_i & PU_N2_dest_ver_addr_n_i	3	I	Differential input. Vertical address in the packet destination. (**VDD_NET and VDD_PU**)
PU_N2_reg_addr_i & PU_N2_reg_addr_n_i	10	I	Differential input. Tag field originally thought as the address of a local memory in each PU. (**VDD_NET and VDD_PU**)
PU_N2_reg_part_i & PU_N2_reg_part_n_i	6	I	Differential input. Additional tag field. (**VDD_NET and VDD_PU**)
PU_N2_data_i & PU_N2_data_n_i	256	I	Differential input. Data carried by the *L2 network* packet.
Path from the *L2 network* node to the PU.			
N2_PU_req_VDD_PU_o	1	O	Four-phase handshaking request signal from the *L2 network* node to its PU. (**VDD_PU**)
N2_PU_req_VDD_NET_o	1	O	Same as N2_PU_req_VDD_PU_o. (**VDD_NET**)
N2_PU_ack_i & N2_PU_ack_n_i	1	I	Differential input. Four-phase handshaking acknowledge signal from the PU to its *L2 network*. (**VDD_NET and VDD_PU**)
N2_PU_is_data_VDD_PU_o	1	O	Indicator that the data content of the packet is data or a command. (**VDD_PU**)
N2_PU_is_data_VDD_NET_o	1	O	Same as N2_PU_is_data_VDD_PU_o. (**VDD_NET**)
N2_PU_reg_addr_VDD_PU_o	10	O	Tag field originally thought as the address of a local memory in each PU. (**VDD_PU**)
N2_PU_reg_addr_VDD_NET_o	10	O	Same as N2_PU_reg_addr_VDD_PU_o. (**VDD_NET**)
N2_PU_reg_part_VDD_PU_o	6	O	Additional tag field. (**VDD_PU**)
N2_PU_reg_part_VDD_NET_o	6	O	Same as N2_PU_reg_part_VDD_PU_o. (**VDD_NET**)
N2_PU_data_VDD_PU_o	256	O	Data carried by the *L2 network* packet. (**VDD_PU**)
N2_PU_data_VDD_NET_o	256	O	Same as N2_PU_data_VDD_PU_o. (**VDD_NET**)
L1 network signals (**All VDD_NET**)			
N1_hor_addr_i	4	I	Hardwired *L1 network* node address.
L1 network signals in the path from Right to Left			
N1_tag_addr_RL_i & N1_tag_addr_RL_o	16	I/O	Tag added to a packet. This tag will stay in the *DDR DRAM PHY*, and will be remapped to the answer to a read command from the DDR before sending the answer back into the *L1 network*. This tag can be used by the PU in any way.
N1_addr_RL_i & N1_addr_RL_o	40	I/O	DDR address to which it is desired to write or from which it is desired to read.
N1_tag_RL_i & N1_tag_RL_o	11	I/O	Additional tag added to a packet. The DDR memory has the capability of receiving a tag with a packet. This tag comes back from the DDR with the answer to a read or write command.

N1_op_RL_i N1_op_RL_o	&	2	I/O	Type of operation carried in the input packet. 00 NOP, 01 READ, 10 READ ANSWER, 11 WRITE
N1_data_RL_i N1_data_RL_o	&	256	I/O	Data carried by the packet.
L1 network signals in the path from Left to Right				
N1_tag_addr_LR_i N1_tag_addr_LR_o	&	16	I/O	Tag added to a packet. This tag will stay in the *DDR DRAM PHY*, and will be remapped to the answer to a read command from the DDR before sending the answer back into the *L1 network*. This tag can be used by the PU in any way.
N1_addr_LR_i N1_addr_LR_o	&	40	I/O	DDR address to which it is desired to write or from which it is desired to read.
N1_tag_LR_i N1_tag_LR_o	&	11	I/O	Additional tag added to a packet. The DDR memory has the capability of receiving a tag with a packet. This tag comes back from the DDR with the answer to a read or write command.
N1_op_LR_i N1_op_LR_o	&	2	I/O	Type of operation carried in the input packet. 00 NOP, 01 READ, 10 READ ANSWER, 11 WRITE
N1_data_LR_i N1_data_LR_o	&	256	I/O	Data carried by the packet.
L2 network signals (**All VDD_NET**)				
N2_enable_PU_i		1	I	Signal that locally enables the connection between the local PU and the *N2 network*. This signal is intended to be hardwired, and it will be hardwired to '1' for the case of the *L2 network* node containing the *coordinating processor*.
N2_rand_i		111	I	Hardwired signal used for fixing ambiguities in the routing when one or more links are down. For each network node a 111 bits random number is sampled and hardwired to this input.
N2_hor_addr_i		4	I	Hardwired signal indicating the *L2 network* node's horizontal address.
N2_ver_addr_i		3	I	Hardwired signal indicating the *L2 network* node's vertical address.
N2_N_link_down_i N2_W_link_down_i N2_E_link_down_i N2_S_link_down_i	& & &	1	I	Hardwired signals that identify which, if any, of the links are down. The links could be down because they are not working or maybe they are down because of a node at the boundary of the grid. (N stands for NORTH, W stands for WEST, E stands for EAST and S stands for SOUTH)
L2 network signals in the path to the NORTH connection				
N2_packet_N_i		1	I	Indicator that a packet is present in the link from the NORTH neighbor to the local network node.
N2_is_data_N_i		1	I	Indicator that the data content of the packet is data or a command.
N2_reg_addr_N_i		10	I	Tag added to the packet. This tag is supposed to address local memory positions in each PU.
N2_reg_part_N_i		6	I	Additional tag added to the packet.

N2_dest_hor_addr_N_i	4	I	Horizontal address of the packet destination.
N2_dest_ver_addr_N_i	3	I	Vertical address of the packet destination.
N2_frac_cnt_N_i	4	I	Packet's *fractional counter*.
N2_time_cnt_N_i	24	I	Packet's *time counter*. When this packet is delivered, this counter represents the delay the packet suffered in being routed to its destination.
N2_data_N_i	256	I	Data carried by the packet.
N2_healthy_N_i	1	I	Indicator of the state of the link connecting the network node to its NORTH neighbor. This signal is received from the NORTH neighbor.
L2 network signals in the path from the NORTH connection			
N2_packet_N_o	1	O	Same as N2_packet_N_i.
N2_is_data_N_o	1	O	Same as N2_is_data_N_i.
N2_reg_addr_N_o	10	O	Same as N2_reg_addr_N_i.
N2_reg_part_N_o	6	O	Same as N2_reg_part_N_i.
N2_dest_hor_addr_N_o	4	O	Same as N2_dest_hor_addr_N_i.
N2_dest_ver_addr_N_o	3	O	Same as N2_dest_ver_addr_N_i.
N2_frac_cnt_N_o	4	O	Same as N2_frac_cnt_N_i.
N2_time_cnt_N_o	24	O	Same N2_time_cnt_N_i.
N2_data_N_o	256	O	Same as N2_data_N_i.
N2_healthy_N_o	1	O	Indicator of the state of the link connecting the NORTH neighbor to the local network node. This signal is sent to the NORTH neighbor.
L2 network signals in the path to the WEST connection			
N2_packet_W_i	1	I	Indicator that a packet is present in the link from the WEST neighbor to the local network node.
N2_is_data_W_i	1	I	Indicator that the data content of the packet is data or a command.
N2_reg_addr_W_i	10	I	Tag added to the packet. This tag is supposed to address local memory positions in each PU.
N2_reg_part_W_i	6	I	Additional tag added to the packet.
N2_dest_hor_addr_W_i	4	I	Horizontal address of the packet destination.
N2_dest_ver_addr_W_i	3	I	Vertical address of the packet destination.
N2_frac_cnt_W_i	4	I	Packet's *fractional counter*.
N2_time_cnt_W_i	24	I	Packet's *time counter*. When this packet is delivered, this counter represents the delay the packet suffered in being routed to its destination.
N2_data_W_i	256	I	Data carried by the packet.
N2_healthy_W_i	1	I	Indicator of the state of the link connecting the network node to its WEST neighbor. This signal is received from the WEST neighbor.
L2 network signals in the path from the WEST connection			
N2_packet_W_o	1	O	Same as N2_packet_W_i.
N2_is_data_W_o	1	O	Same as N2_is_data_W_i.

N2_reg_addr_W_o	10	O	Same as N2_reg_addr_W_i.
N2_reg_part_W_o	6	O	Same as N2_reg_part_W_i.
N2_dest_hor_addr_W_o	4	O	Same as N2_dest_hor_addr_W_i.
N2_dest_ver_addr_W_o	3	O	Same as N2_dest_ver_addr_W_i.
N2_frac_cnt_W_o	4	O	Same as N2_frac_cnt_W_i.
N2_time_cnt_W_o	24	O	Same N2_time_cnt_W_i.
N2_data_W_o	256	O	Same as N2_data_W_i.
N2_healthy_W_o	1	O	Indicator of the state of the link connecting the WEST neighbor to the local network node. This signal is sent to the WEST neighbor.
L2 network signals in the path to the EAST connection			
N2_packet_E_i	1	I	Indicator that a packet is present in the link from the EAST neighbor to the local network node.
N2_is_data_E_i	1	I	Indicator that the data content of the packet is data or a command.
N2_reg_addr_E_i	10	I	Tag added to the packet. This tag is supposed to address local memory positions in each PU.
N2_reg_part_E_i	6	I	Additional tag added to the packet.
N2_dest_hor_addr_E_i	4	I	Horizontal address of the packet destination.
N2_dest_ver_addr_E_i	3	I	Vertical address of the packet destination.
N2_frac_cnt_E_i	4	I	Packet's *fractional counter*.
N2_time_cnt_E_i	24	I	Packet's *time counter*. When this packet is delivered, this counter represents the delay the packet suffered in being routed to its destination.
N2_data_E_i	256	I	Data carried by the packet.
N2_healthy_E_i	1	I	Indicator of the state of the link connecting the network node to its EAST neighbor. This signal is received from the EAST neighbor.
L2 network signals in the path from the EAST connection			
N2_packet_E_o	1	O	Same as N2_packet_E_i.
N2_is_data_E_o	1	O	Same as N2_is_data_E_i.
N2_reg_addr_E_o	10	O	Same as N2_reg_addr_E_i.
N2_reg_part_E_o	6	O	Same as N2_reg_part_E_i.
N2_dest_hor_addr_E_o	4	O	Same as N2_dest_hor_addr_E_i.
N2_dest_ver_addr_E_o	3	O	Same as N2_dest_ver_addr_E_i.
N2_frac_cnt_E_o	4	O	Same as N2_frac_cnt_E_i.
N2_time_cnt_E_o	24	O	Same N2_time_cnt_E_i.
N2_data_E_o	256	O	Same as N2_data_E_i.
N2_healthy_E_o	1	O	Indicator of the state of the link connecting the EAST neighbor to the local network node. This signal is sent to the EAST neighbor.
L2 network signals in the path to the SOUTH connection			
N2_packet_S_i	1	I	Indicator that a packet is present in the link from the SOUTH neighbor to the local network node.

N2_is_data_S_i	1	I	Indicator that the data content of the packet is data or a command.
N2_reg_addr_S_i	10	I	Tag added to the packet. This tag is supposed to address local memory positions in each PU.
N2_reg_part_S_i	6	I	Additional tag added to the packet.
N2_dest_hor_addr_S_i	4	I	Horizontal address of the packet destination.
N2_dest_ver_addr_S_i	3	I	Vertical address of the packet destination.
N2_frac_cnt_S_i	4	I	Packet's *fractional counter*.
N2_time_cnt_S_i	24	I	Packet's *time counter*. When this packet is delivered, this counter represents the delay the packet suffered in being routed to its destination.
N2_data_S_i	256	I	Data carried by the packet.
N2_healthy_S_i	1	I	Indicator of the state of the link connecting the network node to its SOUTH neighbor. This signal is received from the SOUTH neighbor.
L2 network signals in the path from the SOUTH connection			
N2_packet_S_o	1	O	Same as N2_packet_S_i.
N2_is_data_S_o	1	O	Same as N2_is_data_S_i.
N2_reg_addr_S_o	10	O	Same as N2_reg_addr_S_i.
N2_reg_part_S_o	6	O	Same as N2_reg_part_S_i.
N2_dest_hor_addr_S_o	4	O	Same as N2_dest_hor_addr_S_i.
N2_dest_ver_addr_S_o	3	O	Same as N2_dest_ver_addr_S_i.
N2_frac_cnt_S_o	4	O	Same as N2_frac_cnt_S_i.
N2_time_cnt_S_o	24	O	Same N2_time_cnt_S_i.
N2_data_S_o	256	O	Same as N2_data_S_i.
N2_healthy_S_o	1	O	Indicator of the state of the link connecting the SOUTH neighbor to the local network node. This signal is sent to the SOUTH neighbor.

4.3.3 Network Node Programming Capabilities

It is the focus of this section the augmented capabilities added to each of the network nodes. In Table 4.5 the packet format for both *L1* and *L2 networks* was introduced. For the case of the *L2 network* node, signals $*\_is\_data\_*$ were mentioned to identify the information carried by a packet as data or command. For the case of a command, the 256 data bits were divided into four 64-bit pieces, where bits (191

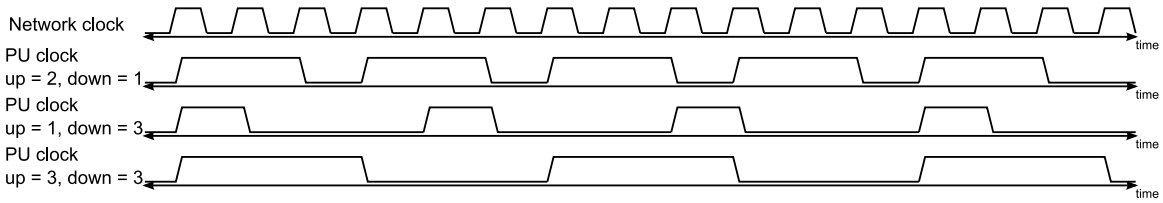


Figure 4.22: Programming the PU clock. Different programmed PU clocks are shown by changing the number of network clock cycles used for when the programmed clock is '0' or '1'.

downto 128) are used in the configuring of the network node.

The clock provided to the PUs through the output signals *PU_clk_VDD_PU_o* and *PU_clk_VDD_NET_o* was mentioned to be programmable. This programming is done through the *L2 network*. A control packet directed to a PU configures its clock if the packet data field is *data(191 downto 184) = "0000001"*. Additionally *data(142 downto 128) = clk_time_up_PU* and *data(157 downto 143) = clk_time_down_PU*, where *clk_time_up_PU* and *clk_time_down_PU* represent the number of clock cycles from the network clock *clk_i* in Table 4.5 that are used for when the programmed clock has to be '1' or '0'. Some of these cases are shown in Figure 4.22. Two extra cases were added to the programming of the local PU clock. The first case is when either *clk_time_down_PU* or *clk_time_up_PU* are all zeros. In this case, the clock provided to the PU is the same $300MHz$ network clock. The second case is when either *clk_time_down_PU* or *clk_time_up_PU* are all ones. In that case the clock supplied to the PU is halted. This was thought to reduce power consumption in PUs that are not being used.

After performing an asynchronous global reset, because that reset signal arrives

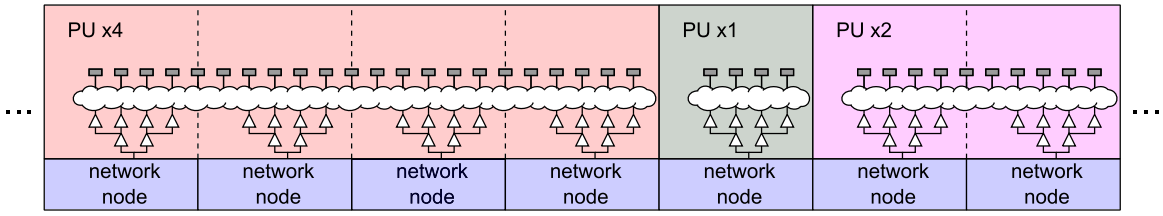


Figure 4.23: Multi-slot PUs. In this example there are three main PUs using the space of seven single PUs. The PUs using more than one slot can use several roots for their clock tree, allowing to achieve better skew and less power dissipation. The grey squares represent registers.

at the same time to all of the network nodes, the local counters responsible of the clock division will be all set to the same starting point. All of these counters will run freely, and all in phase. This characteristic allows that, when two PU clocks are programmed the same way, then they will match not only in frequency, but also in phase. This allows to introduce PUs that could take more than a single PU slot, like in Figure 4.23. These multi-slot PUs will have the capability of using several roots for their clock trees. This technique not only increases the chance of achieving a better clock skew, but it also decreases the clock tree power consumption.

As it was mentioned in Chapter 3, a global reset is provided with a tree to all of the network nodes. This global asynchronous reset is actually forwarded to the PU through the signals *PU_reset_VDD_PU_o* and *PU_reset_VDD_NET_o*. These reset signals were not meant to be used by any type of PU. These signals were provided to the PUs so that PUs containing analog circuitry, such as NVMs, could be set to a known state right away at power-up. For almost all the possible PUs, a global reset is not needed right after power up. For instance, in the case of a purely digital PU,

the reset of the unit could come much later, for instance, after the local PU clock has been set. It is for this reason that the capability of performing a reset in a PU was added through the reception of a reset packet. This gives the flexibility to a PU to change its clock frequency and reset it at any time without the need of power cycling. The command packet performing a reset will have its data field *data(191 downto 184) = "00000001"*. The reset packet has the capability of programming the number of PU clock cycles the reset signal should be kept asserted. This number of clock cycles is set by *data(143 downto 128)*.

Different types of PUs can be incorporated in the CMP designs, and because of this diversity of PUs, it could always happen that one PU design might be faulty. If a PU design is not functioning properly, there is always the risk that PU might inject garbage into any of the networks. It is for this reason that upon power-up, with the assertion of the global reset, all of the PUs are disabled (with the exception of the *coordinating processor* PU), meaning that they are not allowed to send or receive anything to and from the networks. For the case of the reception, packets are received and discarded by the network node. It is only after the reception of an enable packet that a PU is enabled. This packet is identified by *data(191 downto 184) = "00000011"*. Every time that packet is received in a PU, the enable state of that PU is toggled.

Finally, the last control word belongs to the ping packet. It was described before, that upon power-up, a ping packet will be sent to every single network node. A

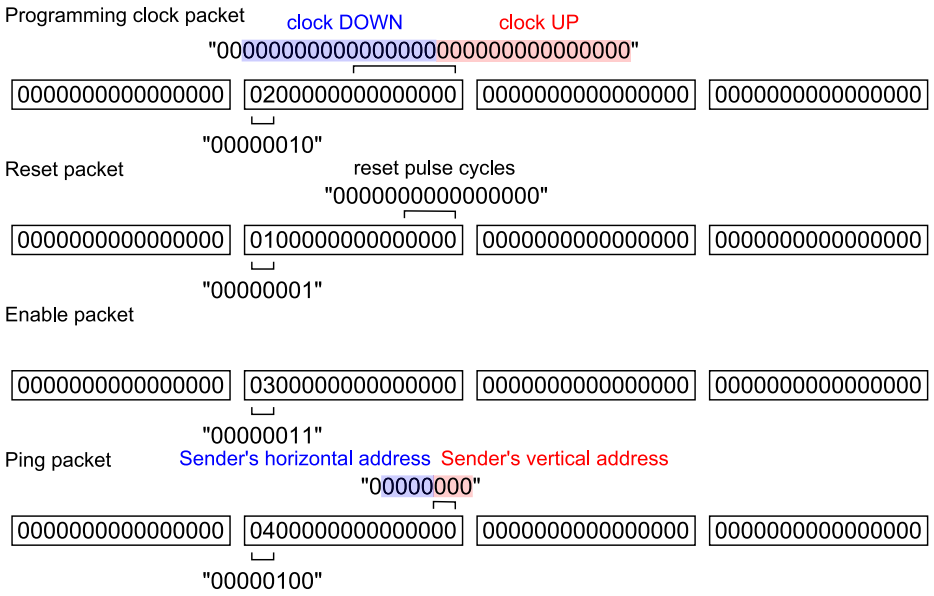


Figure 4.24: Network node's control words. The control words setting the clock, reset and enable signals are shown. The control word responsible of pinging in the network is also shown.

response to every ping is expected to identify the usability of every PU. The ping packet is a control packet with *data(191 downto 184) = "00000100"*. Because the *L2 network* routing is destination based, the origin of the ping packet needs to be added in the control packet. Bits *data(130 downto 128)* will allocate the sender's vertical address, and bits *data(134 downto 131)* will allocate the horizontal one. This ping will be sent by the FPGA interface for instance, and an automatic ping response will be generated by the receiving network node with the original sender's address as its destination. This response will not involve the local PU at all. In Figure 4.24 a summary of the control words is shown.

Chapter 5

Physical Memory Interface DDR

5.1 Introduction

As it was earlier mentioned in Chapter 2, the proposed chiplet solution for an image processing flow would consist of three CMPs mounted on top of an *interposer* chip, along with an FPGA and a 3D DDR memory. The DDR memory is supplied by *Tezzaron Semiconductor*. This memory chip not only features a 3D DDR memory device, but also provides a bridge device that interfaces between the 3D memory and up to four hosts. The four hosts for this project, as seen in 2.2, are three of the chip multi-processors and a Xilinx FPGA. The connection to each of the hosts is done through a 64-bit DDR interface running at a maximum speed of $1.6GHz$. The bridge device supports, on the 3D DDR memory side, the connection to 64 32-bit DDR ports.

This bridge device interfacing with the 3D DDR memory will communicate with the hosts following a certain protocol. Due to the complexity of the protocol, the documentation describing its behavior was not enough for a project of this caliber, and then a *Verilog* model was requested to *Tezzaron Semiconductor*. A very detailed *System Verilog* model of the memory was supplied, allowing to perform close to full-chip logical simulations.

Designing an interface to this external memory posed several challenges, such as the design of custom architectures for delay lines used in the equalization of every bit line coming from the memory, the design of algorithms in performing such delay training, the custom design of clock tree cells as seen in Chapter 3, etc. One of the biggest challenges was actually reaching a speed of $1.25GHz$ in the communication between the memory and CMP. A custom standard cell library was designed with a few characteristics allowing for voltages to be lowered down to $400mV$. Considering additionally the fact that the 55nm process used for the CMPs is a low power one, reaching very high speeds became a very challenging problem.

5.2 *DDR DRAM PHY* Block Division

In dealing with large scale designs, modularity is a must, and then the *DDR DRAM PHY* block was not an exception to the rule. The interface designed to communicate each CMP with the external 3D-DiRAM, occupies an area of $1438.8\mu m$

by 13442.4μm. These are large dimensions, and it is for that reason that the design was split into five main blocks. Figure 5.1 shows the different blocks composing this interface. Two of the blocks shown (DDR_clock_tree and HOST_clock_tree) are not being considered as they represent the clock distributing trees mentioned in Chapter 3. Each of these two blocks would take a 0.8ns clock signal, and would distribute it along both of the longest sides, along with a 2.4ns divided clock. This is done for both the clock signal used in the data flow from a CMP to the DDR memory, and for the received clock used in the opposite direction from the DDR memory to the CMP. The first clock is generated local to each CMP, and it is sent along with data to the 3D-DiRAM. The second clock corresponds to the forwarded version of the first clock by the external memory. This clock is supposed to be more in phase to the received data than the first one.

Block *PAD_interface* is the closest to the CMP's pads communicating with the external 3D-DiRAM. The connection to the external DDR memory is done through two sets of 64 double data rate signals (64 going and 64 coming from the 3D-DiRAM). Making sure that clock signals and the 64 data lines sent to the DDR memory travel the same distance from the CMP to the memory chip, is a very difficult thing to accomplish. It is for this reason that the 3D-DiRAM provided by *Tezzaron Semiconductor* equalizes each of its inputs using a programmable delay. A 16-bit pattern word needs to be transmitted for each of the 64-bit lines going to the memory. This pattern needs to be provided in phase for all of the 64-bit lines. Upon power-up, this

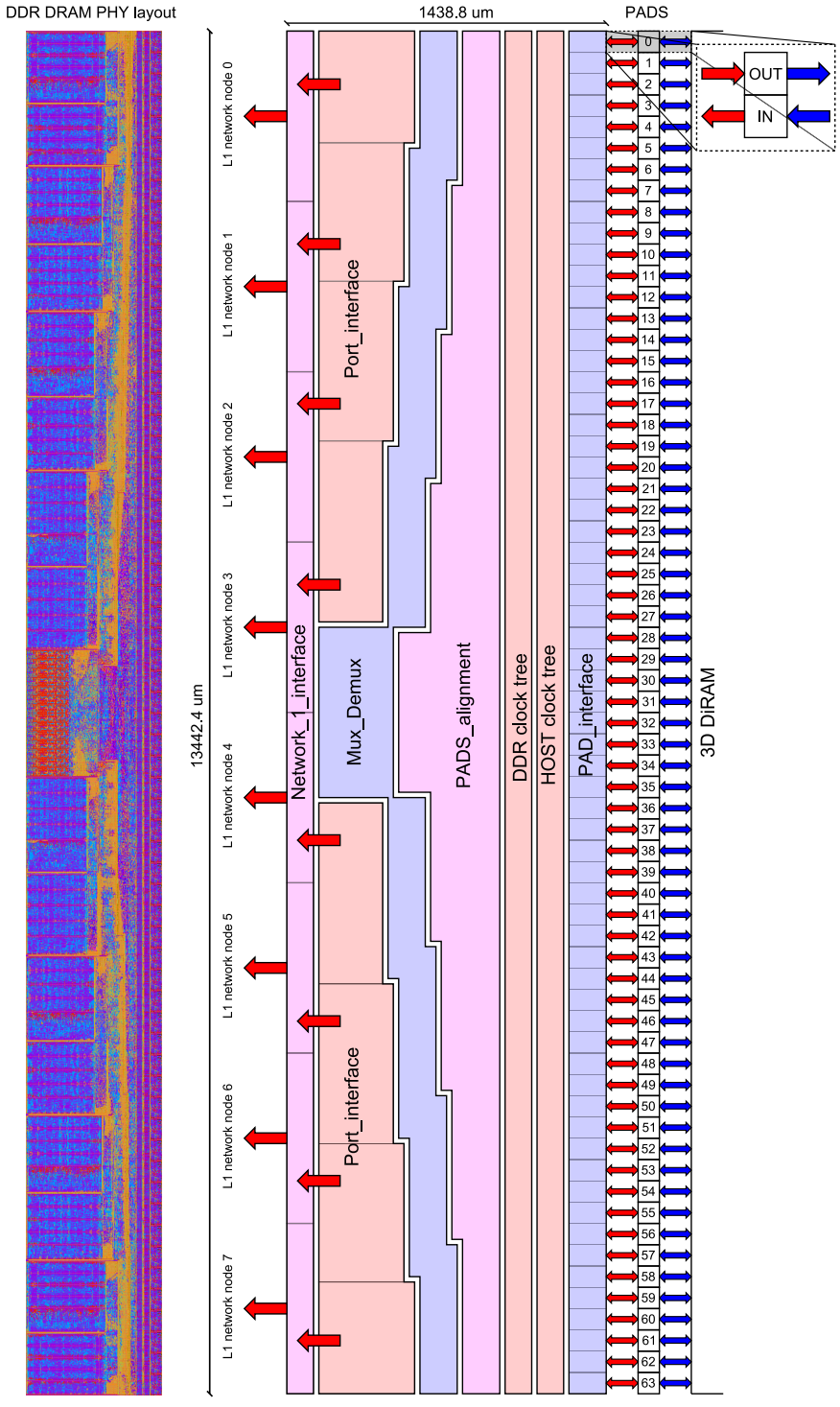


Figure 5.1: *DDR DRAM PHY* hierarchical division. Diagram showing the different blocks composing the *DDR DRAM PHY* block. ($\approx 36M$ transistors)

pattern is sent, and once those programmable delay values are found on the memory side, the same pattern is sent back to the CMP with the objective of performing that same equalization on the CMPs' memory input pads. Block *PAD_interface* is actually divided into 64 smaller blocks, where each of them deals with the transmission of the training pattern and with the local equalization of a single bit line.

As it will be seen later, the innovative programmable delay designed for the CMPs has a minimum delay step of $65ps$, and this delay has not relationship whatsoever with any clock signal. Unfortunately equalizing each of the bit lines coming from the DDR memory using this programmable delay is not enough to ensure same phase among all of the 64-bit lines. It is unlikely, but possible, that due to mismatched delays in the *interposer*, one of the bit lines could be a whole clock period delayed with respect to another line. In this case, in each of the *PAD_interface* individual blocks, the input signal would be equalized correctly, but it is only through the comparison of all of the 64 signal outputs from these blocks that one can assure phase lock. This is the analysis done by the block *PADS_alignment*. After the fine-tune equalization is done local to each of the *PAD_interface* blocks, a second, coarser, equalization is done by the *PADS_alignment* block. This block will perform this equalization using a programmable shift register, where instead of a $65ps$ step, the step will now be the $0.8ns$ clock period.

Error correcting has been incorporated for both packets coming and going from the external 3D-DiRAM. As it can be seen in Figure 5.1, block *PADS_alignment* has a

triangular shape due to the necessity of converging all of the bits in the packet coming from the external memory together, not only because of the second more coarse delay training, but additionally because error correcting needs to be performed on each of the incoming packets, as well as parity bits need to be generated for the out-going packets.

Four big blocks can be identified for each of the *Port_interface* blocks. These are eight blocks, because each of them will communicate to one of the *L1 network* token-rings. Now, considering that read or write commands could come from any of the eight *L1 network* token-rings, and packets coming from the external memory could be redirected to any of those token-ring networks as well, a block performing multiplexing and demultiplexing operation needed to be designed, and this is the *Mux_Demux* block.

Each of the eight smaller blocks in *Port_interface* hold two 48 transaction register files, where incoming read or write commands will be allocated, until the decision is taken to send all of the accumulated commands in a burst to the DDR memory. All of the transactions received by the token-ring networks will be satisfied. If any problem arises and the DDR memory communicates that a transaction could not be performed, this block will keep re-sending those transactions until all of the commands in the register file are successfully executed.

The final big block present in the *DDR DRAM PHY* is the *Network_1_interface*. This block not only allows to distribute the signals coming and going from the *Port-*

_interface evenly in the vertical direction, but it additionally provides augmented capabilities for controlling the traffic to external memory. Remember that because of layout regularity in the PUs and network nodes, the connection to the network nodes from the *Network_1_interface* block needs to take place in specific and evenly spaced places.

5.3 The *PAD_interface* Block

5.3.1 Delay Analysis

The connection from the CMP to the 3D-DiRAM is done through an *interposer* chip (see Figure 2.2). Depending on the physical routing of the *interposer*, each of the signals coming from the 3D-DiRAM can experience different arrival times to the CMP, meaning that the distance traveled by each of the signals could be different. With the objective of correctly latching the values coming from the 3D-DiRAM, data bits traveling to the CMP must satisfy setup and hold times with respect to the used clock. Consequently, just like *Tezzaron Semiconductor* does for its 3D-DiRAM, programmable delays needed to be added to the inputs coming from the DDR memory. These programmable delays will allow a correct reception of the bits coming from the 3D-DiRAM. As it was mentioned before, a training sequence will be sent to the 3D-DiRAM, upon training completion on the memory side, the same sequence will be sent back to the CMP. This 16-bit sequence will be used in configuring

the programmable delays that will be added to each of the inputs coming from the DDR memory. This delay is the one called the *first programmable delay*. Because a whole clock period could happen to be the delay difference between two input signals, a *second programmable delay* had to be added. In this case, this delay corresponds to a programmable shift register, where the minimum delay step is equal to the period of the used clock. This delay will be called the *second programmable delay* and will only implement a delay which is an integer number of clock periods. In Figure 5.2 the three delays mentioned are depicted.

The name d_{prog} will be given to the maximum delay the *first programmable delay* can achieve. Considering symmetry, the maximum time difference between a rising clock edge and the validity of an input signal coming from the 3D-DiRAM can be at most $d_{prog}/2$. An example of how routing distances in the *interposer* can create inconsistency in the arrival time of data from the 3D-DiRAM is shown is Figure 5.3. In red, bits corresponding to the same clock edge are portrayed. The optimum *first programmable delays* are expressed as Δt_0, Δt_1, Δt_2 and Δt_3.

Local to each *PAD_interface* block, even though the maximum delay span can be $[-d_{prog}/2, d_{prog}/2]$, where $d_{prog}/2$ can be higher than P_{clk}, the delay suffered due to the programmable delay line is bounded to $[-P_{clk}/2, P_{clk}/2]$, where P_{clk} is the period of the clock sent to the 3D-DiRAM. The reason for this is that, local to each *PAD_interface* block, a delay equal to the $0.8ns$ clock period would be seen as not existent. For the time being all of the analysis done will consider $d_{prog} > P_{clk}$.

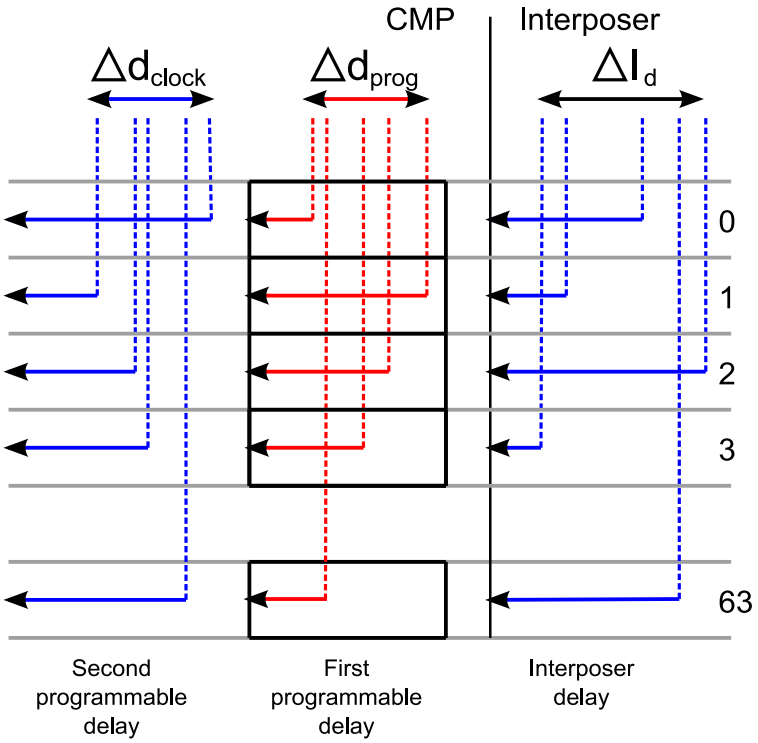


Figure 5.2: Pad equalization delays. An overall depiction of the delays involved in the input pad calibration is presented in this figure. The maximum time difference between two lines in the *interposer* is ΔI_d, the maximum *first programmable delay* is Δd_{prog}, and the maximum *second programmable delay* is Δd_{clock}. The *second programmable delay* will be implemented with a programmable shift register, delaying its input signal in multiples of the clock period.

It can be seen that the correction introduced by the *first programmable delay* does not guarantee that all of the recovered signals coming from the 3D-DiRAM will be in phase, they can actually be delayed by an integer number of clock cycles. Let's assume the difference between the maximum and minimum delays for the input signals due to the routing in the *interposer* is ΔI_d as seen in Figure 5.2. The maximum delay difference that the *first programmable delay* can provide is just one clock cycle (P_{clk}) as long as $d_{prog} > P_{clk}$ holds. The second previously mentioned programmable delay will be implemented using a shift register. The maximum delay that this new stage

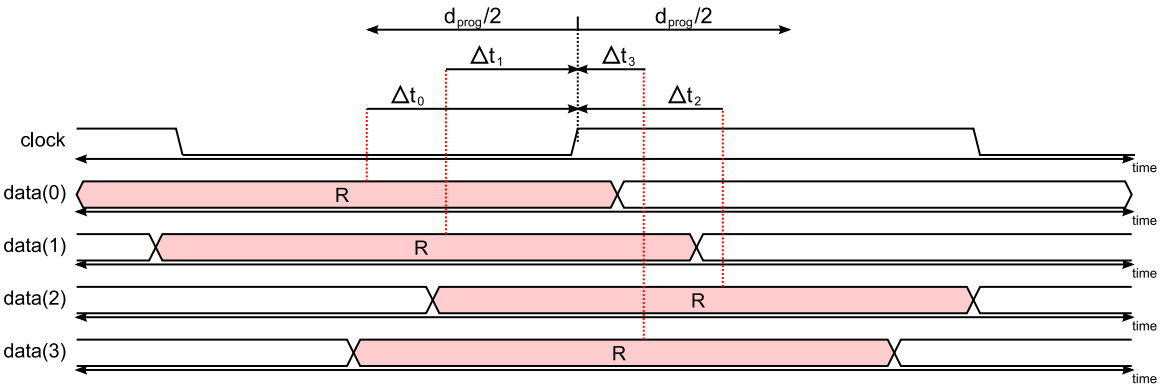


Figure 5.3: ***First programmable delays.*** Example of delays suffered due to the interposer's mismatched distance for the traveling signals. The Δ values are the delays that need to be programmed in the *first programmable delay* lines.

will add is d_{clock}. The training sequence received from the external memory is 16 bits long at double data rate (DDR), with a period of $8.P_{clock}$. The summation of all the maximum difference delays from Figure 5.2 needs to be less than $8.P_{clk}$, otherwise the recovered signals from two input pads could be spaced in time by $8.P_{clk}$. Then:

$$\begin{aligned} & d_{clock} + P_{clk} + \Delta I_d < 8.P_{clk} \\ & \Rightarrow N_{clock} < 7 - \Delta I_d/P, \text{ where } N_{clock} = d_{clock}/P_{clk} \end{aligned} \tag{5.1}$$

So that d_{clock} can be used in the alignment of the different recovered signals, one needs to make sure that d_{clock} is greater than the summation of the remaining delays. Then:

$$d_{clock} > P_{clk} + \Delta I_d \Rightarrow N_{clock} > 1 + \Delta I_d/P_{clk} \tag{5.2}$$

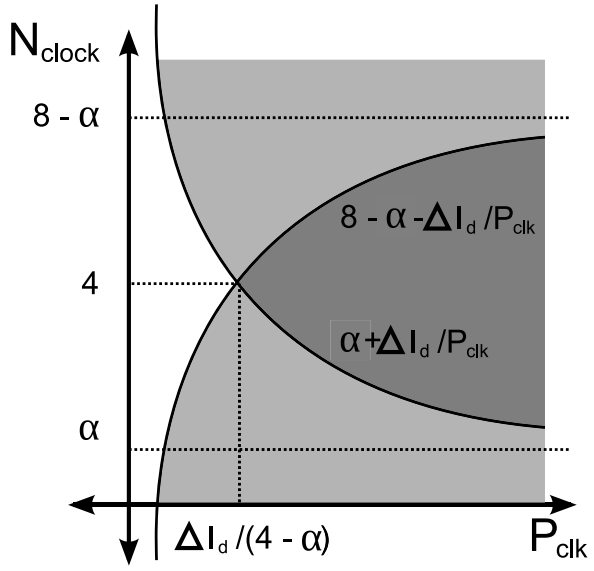


Figure 5.4: Plot of condition in Equation 5.4 The intersection of the two grey areas show the admissible values for P_{clk} and N_{clk}.

The following constraint can be found:

$$7 - \Delta I_d/P_{clk} > N_{clock} > 1 + \Delta I_d/P_{clk}, \text{ where } d_{prog} > P_{clk} \tag{5.3}$$

If the clock period used was such that $d_{prog} < P_{clk}$, then the delay introduced by the *first programmable delay* will be at most d_{prog} and not P_{clk}. Then, considering d_{prog} to be a fraction of P_{clk}, $\alpha.P_{clk} = d_{prog}$, the constraint would be:

$$8 - \alpha - \Delta I_d/P_{clk} > N_{clock} > \alpha + \Delta I_d/P_{clk}, \text{ where } d_{prog} < P_{clk} \tag{5.4}$$

Figure 5.4 plots the two conditions found in Equation 5.4. The intersection of the two gray areas shows the admissible values for P_{clk} and N_{clk}. One can observe that, as long as $P_{clk} > \Delta I_d/(4 - \alpha)$, then $N_{clk} = 4$ is the minimum value to choose.

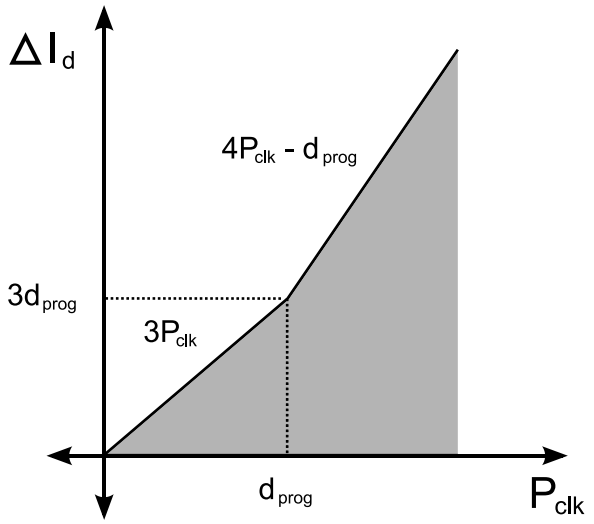


Figure 5.5: Constraints on ΔI_d. The admissible values for ΔI_d are the ones in the shaded region.

A value of 4 was then chosen for N_{clk}, now the maximum value that can be tolerated for ΔI_d needs to be found. Using $\alpha = 1$ when $d_{prog} \geq P_{clk}$, and $\alpha.P_{clk} = d_{prog}$ for when $d_{prog} < P_{clk}$:

$$\frac{\Delta I_d}{4 - \frac{d_{prog}}{P_{clk}}} < P_{clk} \Rightarrow \Delta I_d < 4P_{clk} - d_{prog}, \text{ when } P_{clk} > d_{prog} \tag{5.5}$$

$$\frac{\Delta I_d}{3} < P_{clk} \Rightarrow \Delta I_d < 3P_{clk}, \text{ when } P_{clk} < d_{prog} \tag{5.6}$$

The results found in Equations 5.5 and 5.6 are plotted in Figure 5.5. It can be seen in this figure that as the clock frequency diminishes, a point is reached where $P_{clk} = d_{prog}$, and the constraint on ΔI_d gets more relaxed.

This relaxation is a little misleading, because there is always the requirement that

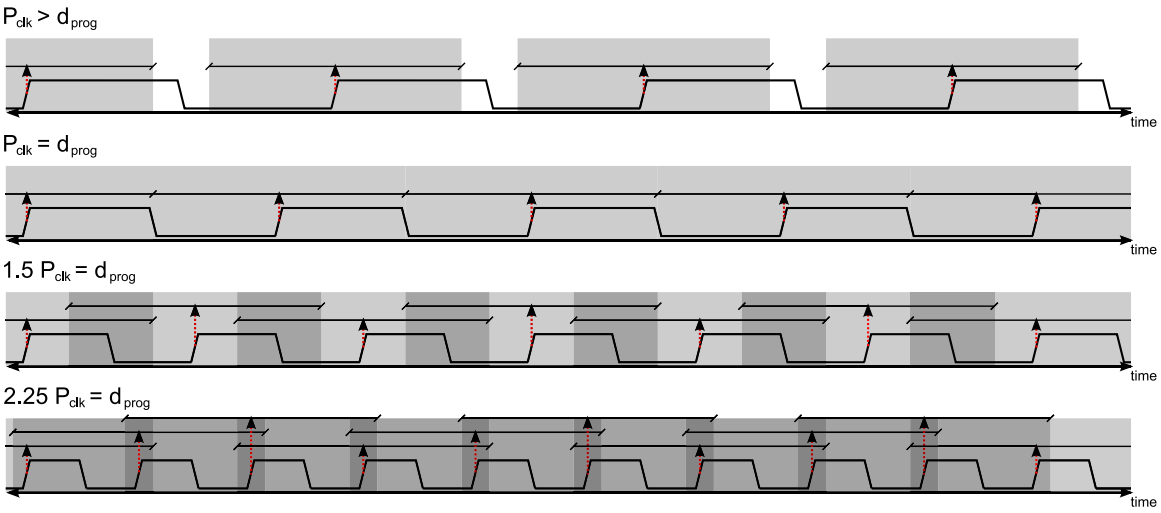


Figure 5.6: *First programmable delay* span for different clock frequencies. Depending on the clock frequency used one can obtain or not overlapping delay span regions.

valid data points need to be within the delay window with respect to the rising edge of the clock. This constraint comes from the *first programmable delay*. Let's take a look at Figure 5.6. In this figure the horizontal lines placed on top of the clock signal are the span of the *first programmable delay*. Some of these spans overlap for the cases where $P_{clk} < d_{prog}$. This condition allows condition in Equation 5.6 to hold, and then the delay difference for two signals coming from the 3D-DiRAM can be at most $3.P_{clk}$. Looking now at the case in which $P_{clk} > d_{prog}$, now the spans of the *first programmable delay* do not overlap. This means that the delays not in the shaded regions cannot be used. In theory Equation 5.5 holds, but there is an additional restriction, that the admissible delays have to be in the shaded region, which might be impossible due to the dependence on the clock frequency used. It is for this reason that Figure 5.5 can be now updated by Figure 5.7, where now for $P_{clk} > d_{prog}$, $\Delta I_d < d_{prog}$.

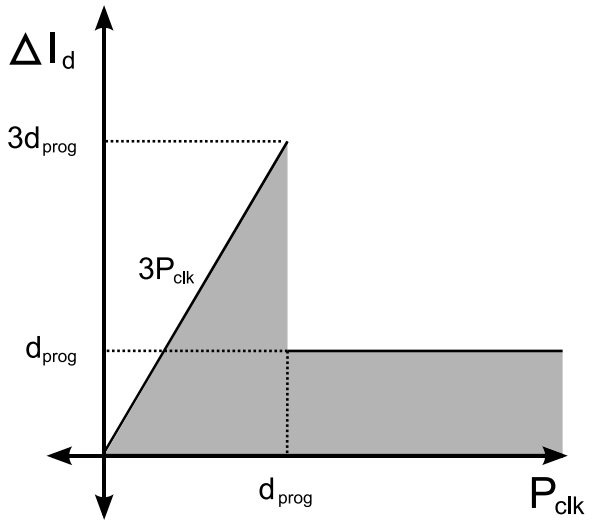


Figure 5.7: Updated constraints on ΔI_d. The admissible values for ΔI_d, as a function of the clock period, are the ones in the shaded region.

5.3.2 Operating Description

The communication between the 3D-DiRAM chip and the *DDR DRAM PHY* will run at double data rate (DDR), and its bus will be 64 bits wide (64 output pins and 64 input pins). Delay calibration for each of these input pins will be done independently of each other, and a dedicated block called *PAD_interface* will receive one of the 64 lines coming from the 3D-DiRAM, and will send data back to the 3D-DiRAM through an output pin. In order to handle the whole bus, 64 of these blocks will then be placed, as seen in 5.1.

Due to the DDR nature of the interface, complementary clocks will be needed. For the case of the data sent to the 3D-DiRAM, one would think that only one clock signal would suffice in the generation of double data rate signals. This is because the state of the clock could be used to multiplex between two single data rate signals in

the generation of a double data rate one. This problem is not as simple as it seems, mainly because when running at high speeds any clock deviation from a 50% duty cycle will unbalance the amount of time used in the transmission of two consecutive bits of information in a single clock cycle, making it harder for the receiver to latch the correct values. An additional problem was found, and that is that CMOS gates react differently to the rising and falling edge of a signal. One could try to equalize this effect by changing the ratio of *pfet* and *nfet* transistors, but this solution unfortunately is not linear with respect to the voltage supply. Because it is one of the objectives in this project to lower power consumption as much as possible, all the voltage supplies are considered to be tunable. A solution needed to be found involving differential clocks, where the rising edge of both negated and not negated clocks are assumed to be spaced in time by half a clock period.

Two complementary clocks will then be used in the data transmission to the 3D-DiRAM, inputs *clkHp_HOST_i* and *clkHn_HOST_i*, and two more clocks will be used in the data reception from the 3D-DiRAM, inputs *clkHp_DDR_i* and *clkHn_DDR_i*. The *HOST clock tree* in Figure 5.1 is the tree cell that provides the *clkHp_HOST_i* clock to each of the *PAD_interface* blocks. The complementary clock *clkHn_HOST_i*, is generated right at the output of the clock tree cell. One could argue that the duty cycle of the output clocks in block *HOST clock tree* could be affected by a change in the supply voltage, and this is true, but in the design of this clock tree cell, very high slew rates were used. This would mitigate the change of the duty

cycle considerably when the power supply is tuned. The complementary clock is then generated by negating the outputs from the *HOST clock tree* blocks. With respect to the two complementary clocks coming from the 3D-DiRAM, maintaining the exact characteristics of these two clocks signals all the way from the external memory to each of the *PAD_interface* blocks, is a very difficult thing to accomplish. It is for this reason that the negated clock, even if it arrives to the CMP, it is not used. Only the positive clock is fed to the *DDR clock tree* block in Figure 5.1. Two clocks will be used when latching the DDR data coming from the 3D-DiRAM, and again the rising edge for both complementary clocks will be used for this. The negated clock will again be locally generated at the outputs of the *DDR clock tree* block. In order to find the time at which this generated complementary clock aligns to its corresponding DDR data bit, the negated clock will be incorporated in the algorithm used to find the correct programmable delays. Two of the *first programmable delays* will be used local to each *PAD_interface* block, where not only the received data bit is delayed, but the negated clock as well.

The 3D-DiRAM has the capability of running at a maximum speed of $1.6GHz$, and then the before mentioned clocks should get as close as possible to that speed. After improving the design several times, the maximum frequency achievable for the differential clocks used in the *DDR DRAM PHY* was $1.25GHz$ ($0.8ns$). Using this frequency for the whole *DDR DRAM PHY* block was a very difficult task, and it is for that reason that two additional clocks are introduced, *clkL_HOST_i* and

clkL_DDR_i. Due to the fact that a write request to external memory and a read answer from external memory both use three *clkHp_HOST_i* clock cycles, the two additional clocks presented will run at three times slower speed. Meeting timing constraints for a $2.4ns$ clock period instead of a $0.8ns$ one, made the design of the *DDR DRAM PHY* simpler. This is the main reason two in phase $2.4ns$ and $0.8ns$ clocks were shown in Figure 3.5. The whole *DDR DRAM PHY* will be divided into two clock domains, one using *HOST* clocks, which are the ones used in the flow from the CMP going to the 3D-DiRAM, and *DDR* clocks, which are the ones used for the opposite flow direction from the 3D-DiRAM to the CMP.

Upon startup, the 3D-DiRAM will expect the reception of a training sequence in order to find the correct programmable delays for its own input pads. When this training process has concluded, this same training sequence will be sent back to the CMP, so that training can be executed on the CMP's input ports. A local copy of the expected training sequence will be kept on each of the input pad training circuitry. The comparison between the local copy of the training sequence present in each *PAD_interface* block, and the one received from the 3D-DiRAM, is performed while changing the programmable delays applied to the data input pin, and additionally by circularly shifting the local train sequence copy. Because of the local generation of the *clkHn_DDR_i* negated clock, the training of the negated clock input was incorporated. The way coincidence between the two sequences is found, is by changing delays in the way depicted in Figure 5.8. The *first programmable delays* are tested in the following

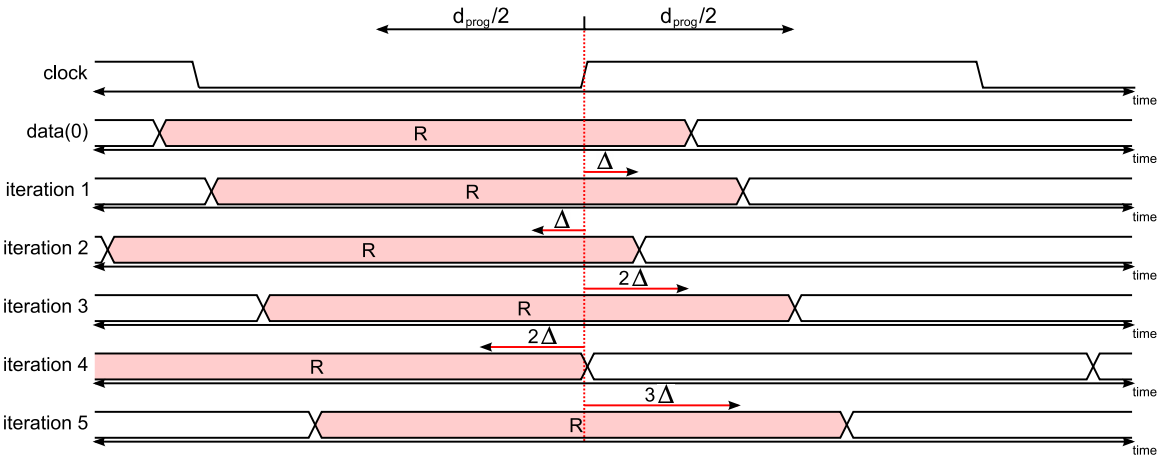


Figure 5.8: Search for the optimum *first programmable delay.* The way in which the *first programmable delays* are tested is shown in this figure. Using this procedure, the lowest added delay to the input signal can be achieve. This added or subtracted delay can be at most half of the clock period $P_{clk}/2$. The maximum time difference among all of the input signals will be at most a clock period P_{clk}.

order Δt, $-\Delta t$, $2\Delta t$, $-2\Delta t$, etc for the data input. This way of testing delays assures that the minimum absolute delay value is added or subtracted to the signal. For the negated clock input, the change in delays is done linearly.

The 64 *PAD_interface* blocks that will independently train each of the input pins are presented at the bottom of Figure 5.9. Each of them is supplied with the six mentioned clocks.

Because the *PAD_interface* block is being synthesized by itself, and will afterwards be instantiated on a higher level as a block, for timing purposes, all of the inputs and outputs were decided to be registered. This prevented from any combinatorial logic to be placed in front of the input pin registers, or at the output of the output pin registers, alleviating input/output delay problems on the higher level of hierarchy. The input and output signals that contain the word *HOST* are driven by the *HOST*

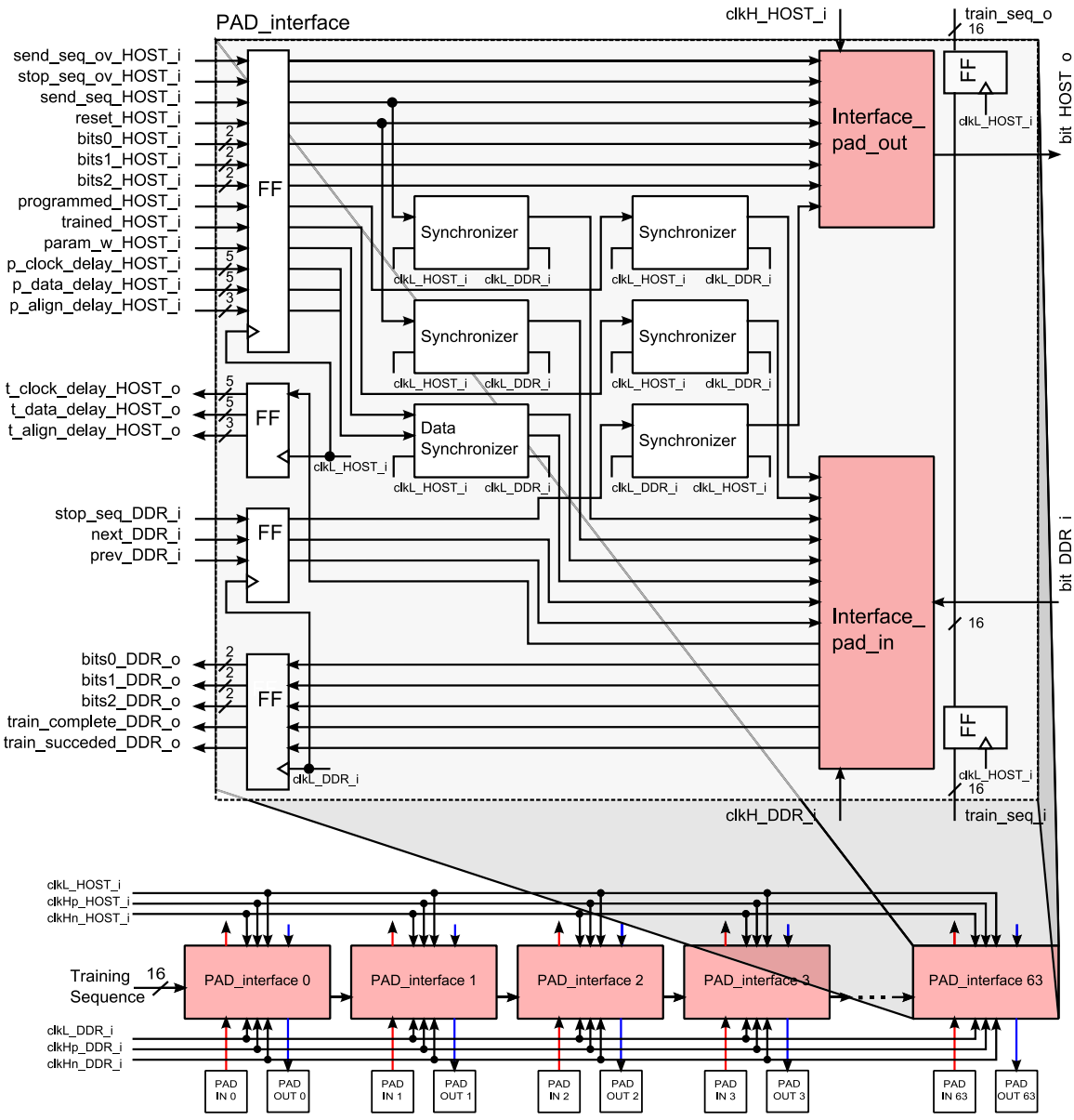


Figure 5.9: *PAD_interface* block. General structure for the *PAD_interface* block.

clocks, and the ones with *DDR* are driven by the *DDR* clocks. The general idea of how this block works is the following:

1. A reset pulse is received through *reset_HOST_i* for setting the system at a known state.

2. A pulse through *send_seq_HOST_i* will trigger the training process. This pulse will indicate the block *Interface_pad_out* to start sending the training sequence to the 3D-DiRAM and, at the same time, it indicates the block *Interface_pad_in* that the training sequence will be received once the input pads from 3D-DiRAM have finalized with their training. The block *Interface_pad_in* receives this indicator by crossing the *HOST* clock domain to the *DDR* clock domain. This is done with the block *Synchronizer*.

3. Once the training for the *first programmable delay* has finished, the training could have succeeded or not. The output *train_complete_DDR_o* will indicate that the training has finished, either because the correct delay values have been found, or because all of the combinations have been tried and none of them worked. To determine if the training succeeded, output signal *train_succeeded_DDR_o* will indicate so. If this signal is asserted along with the *train_complete_DDR_o* signal, the training was successful, otherwise it was not.

4. Either with a successful training or not, a pulse will be sent to the input signal *stop_seq_DDR_i*, stopping the training sequence from being transmitted to the

3D-DiRAM, and consequently no training sequence will be received back from the external memory.

5. In case the training was unsuccessful, one can always read the trained delays for both data input and clock input through the signals *t_clock_delay_HOST_o* and *t_data_delay_HOST_o*. If any kind of problem is identified with these trained values, one can manually program them. First, if needed, by using the input signals *send_seq_ov_HOST_i* and *stop_seq_ov_HOST_i*, one can manually start and stop the transmission of the training sequence to the 3D-DiRAM so that training can be done on the external memory input pads. Once this training finishes, using the input signal *param_w_HOST_i* as a write enable signal, one can write the delay values for the data input and negated clock input through *p_data_delay_HOST_i* and *p_clock_delay_HOST_i*. If one decides to switch between the programmed or trained delay values for the *first programmable delay*, a pulse should be sent to one of the input signals, *trained_HOST_i* or *programmed_HOST_i*.

6. Finally the *second programmable delay* needs to be calculated. This *second programmable delay* can be written through the signal *p_align_delay_HOST_i* using the write enable signal *param_w_HOST_i*. The choice for values in this *second programmable delay* will depend on all of the *PAD_interface* blocks and the values chosen for their *first programmable delay*, and it will also depend on

the algorithm used for calculating them. Just like for the *first programmable delays*, the *second programmable delay* trained values can be read with the output *t_align_delay_HOST_o*. With a system reset, the *second programmable delay* is set to zero and the input of the first register of the four stages shift register is chosen. By receiving pulses through the inputs *next_DDR_i* and *prev_DDR_i*, the position in that four stages shift register can be increased or decreased.

Figure 5.10 shows the architecture for the *second programmable delay*. It can be seen in this figure that the input data coming from the 3D-DiRAM and the negated clock are passing through two *first programmed delays*. The first two registers save the value of the input signal *bit_DDR_i* at the rising edge of the two complementary clocks. After this, the shaded region shows the four stages shift registers used for the *second programmable delay* (shift register size chosen base on Figure 5.4, where N_{clock} was decided to be 4). There are two shift registers because one belongs to the data latched with the positive clock and the other one corresponds to the negative clock. A counter driven by signals *prev_DDR_i* and *next_DDR_i* will select the position in both shift registers. After this stage an additional counter will drive the signals *S1*, *S2* and *S3*, allowing to convert the two outputs from the multiplexers into six slower streams running at three times slower clock frequency (*clkL_DDR_i* at $2.4ns$).

The algorithm used for the training of the *first programmable delay* will be now introduced. The cell responsible of performing the variable delay is called *SEN_DELAY*

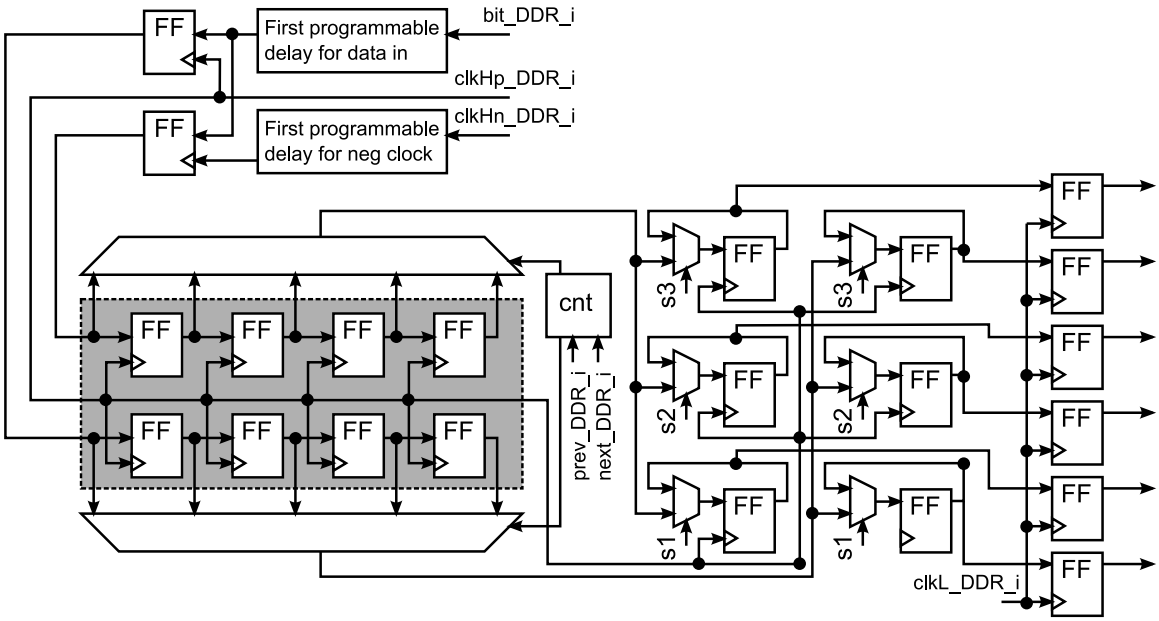


Figure 5.10: ***First programmable delay*** **+** ***second programmable delay*** **+ data rate conversion.** Diagram showing the *first programmable delay*, the architecture of the *second programmable delay*, and the data rate conversion from a DDR $0.8ns$ clock period, to six SDR $2.4ns$ clock period signals. The *second programmable delay* is shown in the shaded region, and the data rate converting unit is shown starting from the placement of the two big multiplexers on top and bottom of the *second programmable delay*.

and it will be introduced later in the chapter. This delay cell has two inputs and one output. The two inputs correspond to the input desired to be delayed, and the input configuring that delay. This last signal is a 64-bit signal where only one bit is allowed to be '1'. A shift to the right from the MSB to the LSB of that '1' will result in a monotonically decreasing delay at the only cell output. Two of these delays will be used as seen in Figure 5.10. The signal controlling the delay in the data bit will be called *current_delay_data*, and the one for the negative clock *current_delay_clock*. When training starts *current_delay_clock* will begin with value "x8000000000000000" and *current_delay_data* will begin at "x0000000080000000". This means that the

first signal will start with the maximum programmable delay, and the second signal will start at half the maximum delay. For every set of *current_delay_data* and *current_delay_clock* values, 256 0.8ns clock periods will be used to test if for at least 128 consecutive clock periods, the input signal matches with the expected local running training sequence. If after those 256 clock cycles the training sequence has not been locked, an additional shift is performed on the local running expected training sequence. This keeps happening up to 15 shifts (because the training sequence is 16 bits). After the 15th shift, a shift in the *current_delay_data* signal is applied. This shift is perform following the *STAGE 1* configuration of the *current_delay_data* register seen in Figure 5.11. The shown shift will perform the following sequence of shifts 1, -1, 2, -2, 3, etc. This allows the closest to the half maximum delay to be chosen as a valid delay for signal *current_delay_data*. If all of the possible shifts have been tested for *current_delay_data*, then a shift to the right is applied to the clock delay signal *current_delay_clock*. If all of the possible combinations are tested and no sequence lock was found, the training is considered to have failed. If on the other hand, the sequence happens to lock, then the *current_delay_data* register changes its configuration from *STAGE 1* to *STAGE 2* in Figure 5.11. If the '1' in *current_delay_data* signal is found to be between the 63 and the 32 bit, then the next shifts applied to signal *current_delay_data* will be to the left, otherwise the shift will be done to the right. The shift will continue happening until the sequence lock has been lost. If that happens, then the midpoint between the first *current_delay_data* configuration that

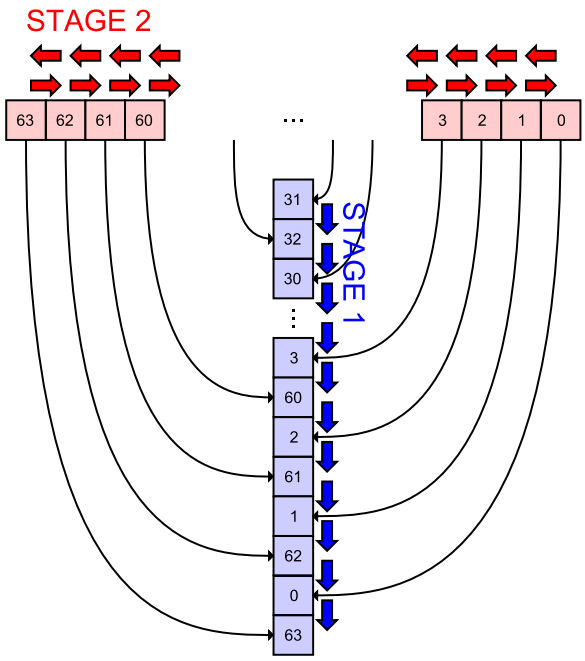


Figure 5.11: Shift-register controlling the *first programmable delay* applied to one of the 64 signals coming from the 3D-DiRAM. Two configurations are used for this register. STAGE 1 will be used until the training sequence has been locally locked, and STAGE 2 will be used after the sequence was locked until it is lost.

locked the sequence, and the last one is performed. The value obtained in this way will be the most likely to work at all times, as it will belong to the center of the eye diagram for the incoming bit line.

Figure 5.12 and 5.13 shows the architecture of the block that allows the transmission of a pulse from one clock domain to another, and the block that allows to transmit data from a bus from one clock domain to another. For the case of the *Synchronizer* block, the registers can start at any value. If a '1' is found in any of them upon power-up, as long as the input *in1_i* is not asserted, these ones will be flushed through the output *out2_o*. The input *in1_i* needs to be asserted for only one

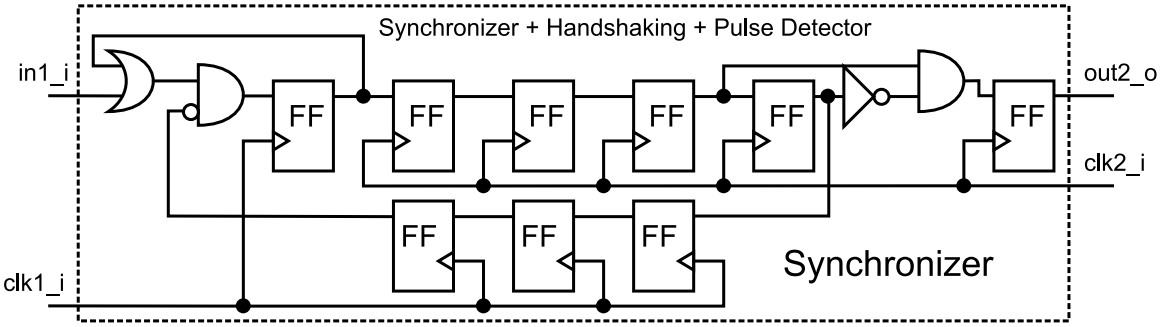


Figure 5.12: Synchronizer used for clock domain crossing. Architecture used for passing from one clock domain to another. There is no restriction on the clock frequencies.

clock cycle, and after a while that pulse is sent out through the output *out2_o*. If an acknowledge was needed on the input side, the output of the last register can be used as such. For the case of the *Data_synchronizer* block, the principle is the same. An enable signal is pulsed along with valid input data, and after a few clock cycles, a pulse is generated for the output in the other clock domain. When the output *en_o* pulses, the data present in *data_o* is valid.

5.3.3 Programmable Delay Cell *SEN_DELAY*

Several options were considered for a programmable delay architecture. Unfortunately, all of the analyzed options either relied on a fine and coarse step programmability,[28–31] or resistors were used for the delay calibration,[32] or special biasing had to be considered.[33] In the designed delay line that will be presented here, not only monotonically increasing delays are achieved without the need of a coarse and fine calibration, but also no biases or special cells are required. A very compact, CMOS-

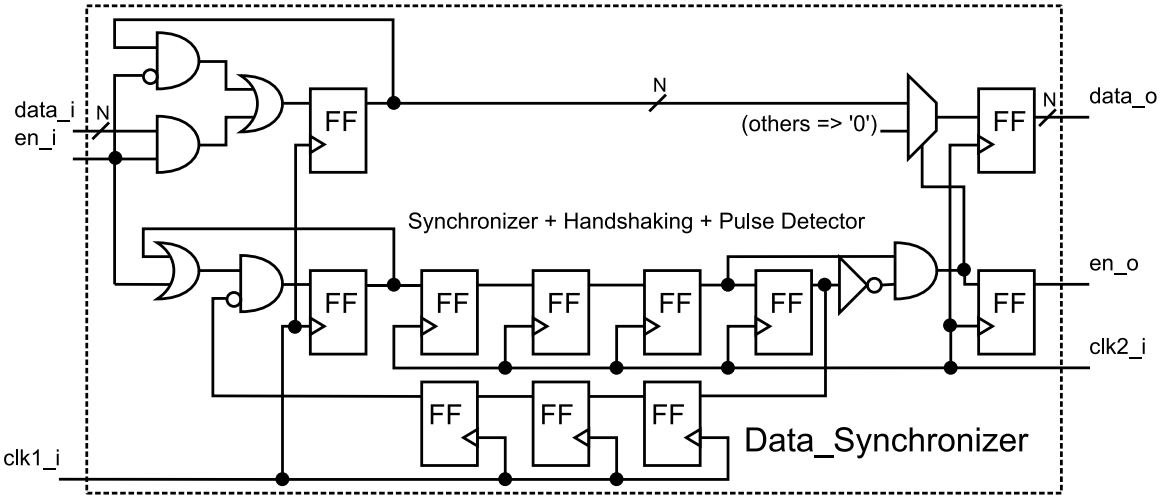


Figure 5.13: Synchronizer used for clock domain crossing when data is transmitted. Architecture used for passing data from a bus from one clock domain to another. There is no restriction on the clock frequencies.

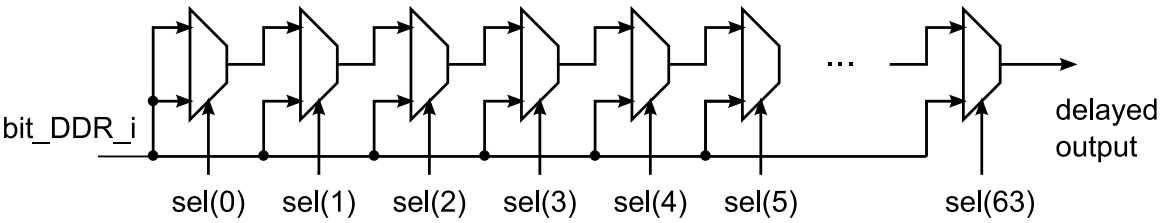


Figure 5.14: *First programmable delay* architecture. Architecture used for setting the two *first programmable delays* corresponding to inputs *bit_DDR_i* and *clkHn_DDR_i*.

scalable design, with a delay step of just $65ps$ is presented here for the used $55nm$ GF process.

Figure 5.14 shows a possible architecture in the delay for both the negated clock and the data signal coming from the 3D-DiRAM. In this figure, starting from right to left, the first multiplexer that allows the input signal *bit_DDR_i* to pass through it, will be the one determining the delay. Several number of steps were tried, starting from 16 to 64, and several designs for the multiplexer unit have been explored.

A two input CMOS multiplexer provides a negated output, and then this output needs to be additionally inverted if the multiplexing units presented in Figure 5.14 are desired to be obtained. This can be done by just using an inverter, but if one considers $f_r(t)$ and $f_f(t)$ two functions that characterize the rising transition and falling transition at the output of the two input CMOS MUX, it is important that $f_r(t) = f_f^{-1}(t)$. If this condition doesn't hold, after passing through several of these multiplexing stages, the duty cycle of the input signal will not be maintained. In order to solve this problem, the first approach taken is the one presented in Figure 5.15a. By having a second multiplexer at the output of the first one, $f_r(t)$ and $f_f^{-1}(t)$ are more similar. The problem here is that the second input of the second multiplexer will be set to either '0' or '1', and then there is no way to replicate the same inputs for this second multiplexer, making the matching of the rising and falling functions difficult. An additional architecture was designed, which was found to be very successful in terms of maintaining the duty cycle of a signal through almost hundreds of stages. This is the case of Figure 5.15b. In this case symmetry has been exploited and capacitances are being matched, making $f_r(t) \approx f_f^{-1}(t)$. The transistor-level architecture for option 5.15b is presented in 5.15c. It can be observed that the first column of multiplexers in 5.15b have the same two inputs but swapped, so both cases are being considered at the same time. The second column of multiplexers, regardless of the value the input signal S_i may have, they will also have the same inputs as the first column of multiplexers, but in this case inverted. If this architecture is used for Figure 5.14, it

can be seen that all of the nets will have the same capacitance, because the output of a two input CMOS multiplexer will always go to two inputs. This approach was very successful in terms of maintaining the duty cycle of the input signal, but the minimum step delay was found to be too much for the clock speed used in the *DDR DRAM PHY*. The delay achieved for a single stage would be around $110ps$, which only gives us roughly 7-8 points for a $800ps$ clock period ($1.25GHz$). A possible solution that involved using this architecture could have been using this unit for performing a coarse delay training, and then use an additional delay line that would use the architecture in Figure 5.15a for fine tuning. The problem with this option is that monotonically increasing delays cannot be ensured. This is particularly important because the algorithm used to calculate the delays, calculates the average of two points as seen in the previous section. This calculation would become obsolete if monotonically increasing delays cannot be warrantied.

A better architecture was found, one that does not maintain the duty cycle as well as the one presented before, but for up to 100 delay stages, the change in it is minimum. For this new architecture, a single step delay of $65ps$ was achieved, which now gives almost 13 points for a period of $800ps$. Figure 5.16 shows the general architecture of this new design, and in Figure 5.17 the architecture for the single delay is presented. Comparing the architecture in Figure 5.16 to 5.14, it can be observed there are two signals being carried along the way instead of just one. These signals are the ones named P and P_n. The reason to carry two signals is that they

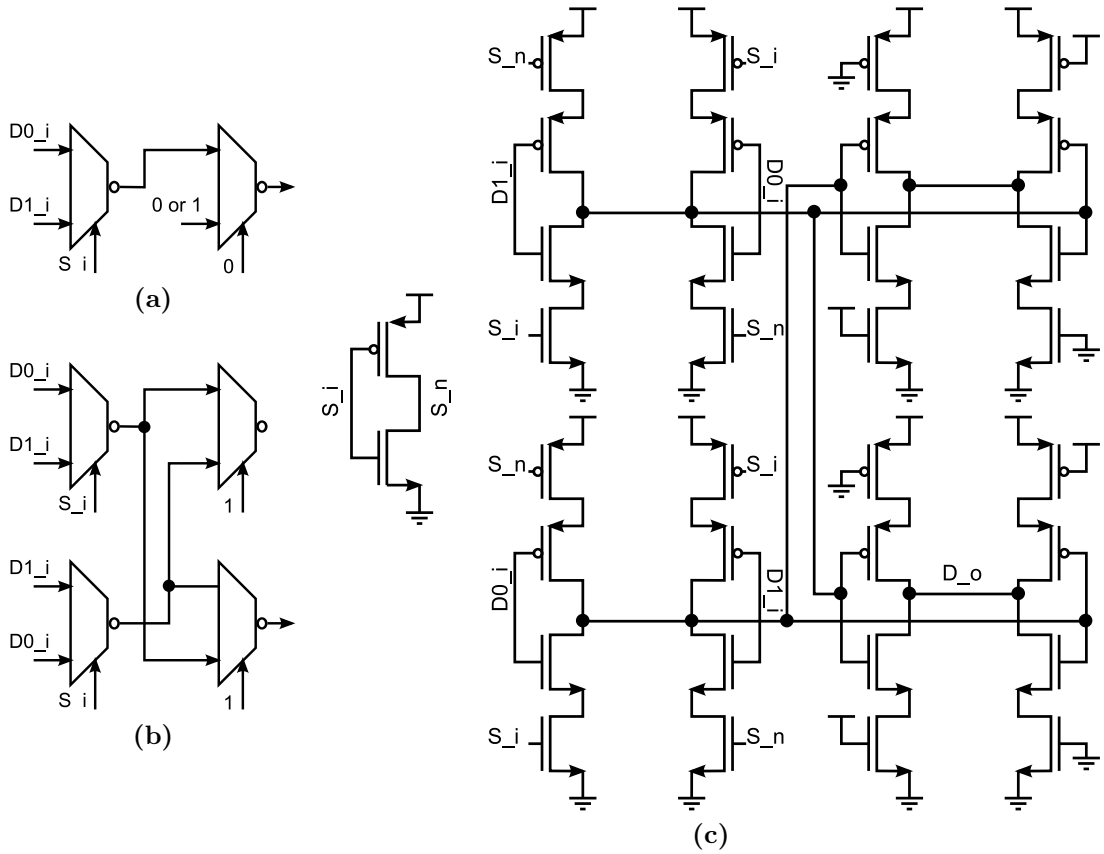


Figure 5.15: Multiplexer approaches. Three different approaches for the multiplexer used as the minimum step delay for the *first programmable delay*.

are differential. Carrying a differential signal allows not to need two inverting stages like in Figure 5.15a or 5.15b, reducing the delay significantly. In Figure 5.17 the control signals for the full transmission gates can be considered static, because once the desired delay has been programmed, they are never changed again. The static control inputs of the full transmission gates allow them to transmit their input to output very quickly. Additionally, in the two inverter-like structures in 5.17, it can be seen that the output of one of them will not switch until the input of the other has switched. This allows the differential signal to be auto-regulated, so that phase

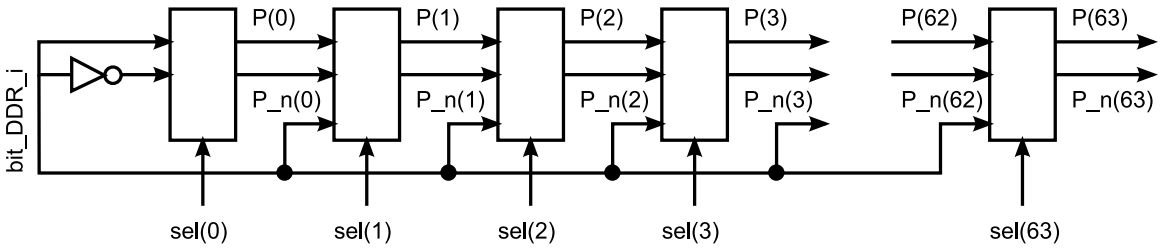


Figure 5.16: New *first programmable delay* architecture. Architecture used for the data input *bit_DDR_i* and the negated clock *clkHn_DDR_i*.

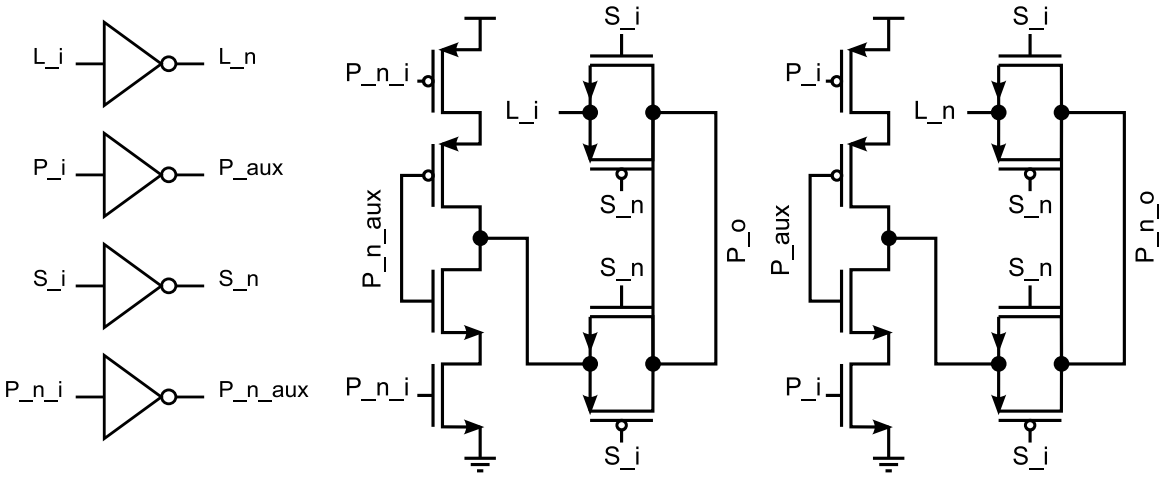


Figure 5.17: Single delay architecture. Architecture for the single delay used in the *first programmable delay* used in the data input *bit_DDR_i* and the negated clock *clkHn_DDR_i*.

can be kept for all of the delay stages. The area used for each of these 64-stage programmable delay unit is $2131.2\mu m^2$. For an input clock of $1.25GHz$, if the signal is passed through all of the delaying elements, only a 4% duty cycle change is suffered, and a maximum power dissipation of $5.7mW$.

5.3.4 DDR Output Generator Cell *SEN_DDR*

As mentioned before, a differential clock is used in the transmission from the *DDR DRAM PHY* to the 3D-DiRAM. Due to the fact that the clock signals might not be 50% duty cycle, a single clock is not recommended to be used to send data in both high and low states of the clock. The assumption made for this cell is that the rising transition for both clocks happen at the correct time. A problematic case is the one where both the positive and negative clocks are overlapping for a period of time, and then a solution involving the identification of the rising edge transitions in both clocks needed to be found. A solution to this problem is found in Figure 5.18. This circuit uses the rising transition of both clocks to generate a differential clock signal with signals *QP* and *QN*. When a rising transition is recorded for *clkHp_HOST_i*, *QP* will go to '0' and *QN* will go to '1', additionally a clear signal will set to '0' signal *P*. When a rising transition happens for the other *clkHp_HOST_i* clock, the opposite happens, and a clear signal will set *N* to '0'. The idea is that these two signals *QP* and *QN* can be used in the output multiplexer to transmit either the bit from input *d0_i* or *d1_i*. The four additional registers for inputs *d0_i* and *d1_i*, and the delaying buffers will prevent output *d_o* from generating glitches. A timing diagram is presented is Figure 5.19. Any initial state for the registers and the RS flip-flop will eventually converge to the behavior seen in 5.19.

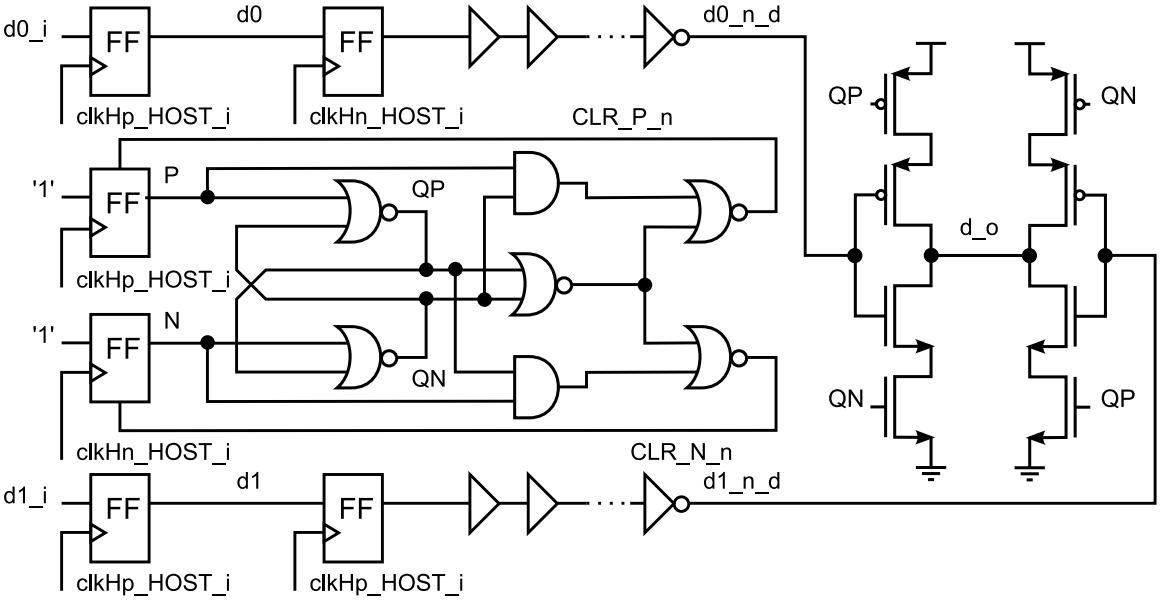


Figure 5.18: The *SEN_DDR* cell. Double data rate cell. This cell takes two input bits and a differential clock input, and generates a DDR output with 50% duty cycle. The two clock signals do not need to be non-overlapping, and the duty cycle doesn't need to be 50% either.

5.3.5 Input/Output Signals

A brief description of what each of the input/output signals from block *PAD-_interface* are presented in Table 5.1.

Table 5.1: Description of the *PAD_interface* signals.

Signal name	Bits	O/I	Description
train_seq_i & train_seq_o	16	I/O	The input port is the input supplying the expected training sequence. The output port corresponds to the training sequence that is supplied to the following *PAD_interface* block from the input train_seq_i.
Clock domain going from the CMP to the 3D-DiRAM			
clkHp_HOST_i & clkHn_HOST_i	1	I	Differential high frequency clock (0.8ps period clock) used for the path going to the 3D-DiRAM.
clkL_HOST_i	1	I	Low frequency clock (2.4ns period clock) used for the path going to the 3D-DiRAM.

send_seq_ov_HOST_i	1	I	With a pulse of this signal, the output pad interface block starts sending the training sequence to the 3D-DiRAM without performing the training in the input pad interface block. This is mainly intended for the case in which manual configuration of the programmable delays is performed.
stop_seq_ov_HOST_i	1	I	If manual configuration of the programmable delays is performed, a pulse was sent to the send_seq_ov_HOST_i input so that the 3D-DiRAM can train its input pads. The input stop_seq_ov_HOST_i puts a stop to the transmission of the training sequence.
send_seq_HOST_i	1	I	A pulse will indicate that the training sequence needs to be sent. Once the 3D-DiRAM pads have been trained and the training sequence is received back from the external memory, the CMP input pads start their training.
reset_HOST_i	1	I	Reset request. Upon power up, a reset pulse needs to be received.
bits0_HOST_i & bits1_HOST_i & bits2_HOST_i	2	I	Three pair of data bits needed to be sent to the 3D-DiRAM using the clkHp_HOST_i clock. In $2.4ns$ six parallel bits will be translated sequentially every $0.4ns$.
programmed_HOST_i & trained_HOST_i	1	I	A pulse in programmed_HOST_i will select the manually programmed delay values. On the other hand, a pulse in trained_HOST_i will select the values obtained through the training.
t_clock_delay_HOST_o	6	O	Achieved delay for the clock signal through training. (*first programmable delay*)
t_data_delay_HOST_o	6	O	Achieved delay for the data signal through training. (*first programmable delay*)
t_align_HOST_o	4	O	Achieved delay for the data signal through training. (*second programmable delay*)
param_we_HOST_i	1	I	Write enable signal for the inputs p_clock_delay_HOST_i, p_data_delay_HOST_i and p_align_HOST_i.
p_clock_delay_HOST_i	6	I	Same as t_clock_delay_HOST_o, but in this case this is the manually chosen delay.
p_data_delay_HOST_i	6	I	Same as t_data_delay_HOST_o, but in this case this is the manually chosen delay.
p_align_HOST_i	4	I	Same as t_align_HOST_o, but in this case this is the manually chosen delay.
bit_HOST_o	1	O	Double data rate ouput bit line going to the 3D-DiRAM.
Clock domain going from the 3D-DiRAM to the CMP			
clkHp_DDR_i & clkHn_DDR_i	1	I	Differential high frequency clock ($0.8ps$ period clock) used for the path coming from the 3D-DiRAM.
clkL_DDR_i	1	I	Low frequency clock ($2.4ns$ period clock) used for the path coming to the 3D-DiRAM.

bit_DDR_i	1	I	Double data rate input bit line coming from the 3D-DiRAM.
bits0_DDR_o & bits1_DDR_o & bits2_DDR_o	2	O	Three pair of data bits are recovered from the six data bits obtained during $2.4ns$ from the input bit_DDR_i.
train_complete_DDR_o	1	O	Output signaling that the training procedure has finished.
train_succeeded_DDR_o	1	O	Output signaling if the training procedure has been successful or not. This signal has to be read once $train\_complete\_DDR\_o =$ '1'.
stop_seq_DDR_i	1	I	When the training sequence needs to stop being transmitted, a pulse to this input is sent.
next_DDR_i & prev_DDR_i	1	I	These inputs are used for controlling the *second programmable delay*. A pulse through next_DDR_i will increase by one clock cycle ($0.8ns$ period) the delay introduced by the *second programmable delay*. A pulse through the prev_DDR_i will decrease it by one clock cycle ($0.8ns$ period).

5.4 The *PADS_alignment* Block

5.4.1 Operating Description

The format for the packets coming and going from the 3D-DiRAM is presented in Table 5.2. Packets are divided in two sizes, 128 bits or 384 bits. The packets carrying data read from memory, or data to be written to memory, will be the 384 bits one, and will carry 256 bits of information. *Command* or *Data* segments will be received from the external memory every $0.4ns$, in groups of 64 bits. Block *PAD_interface* will translate six serial bits from a data bit signal coming from the external memory, into six parallel bits running at three times slower frequency was shown. The generation of these six parallel bits is shown in Figure 5.20 for the case of one of the 64 lines

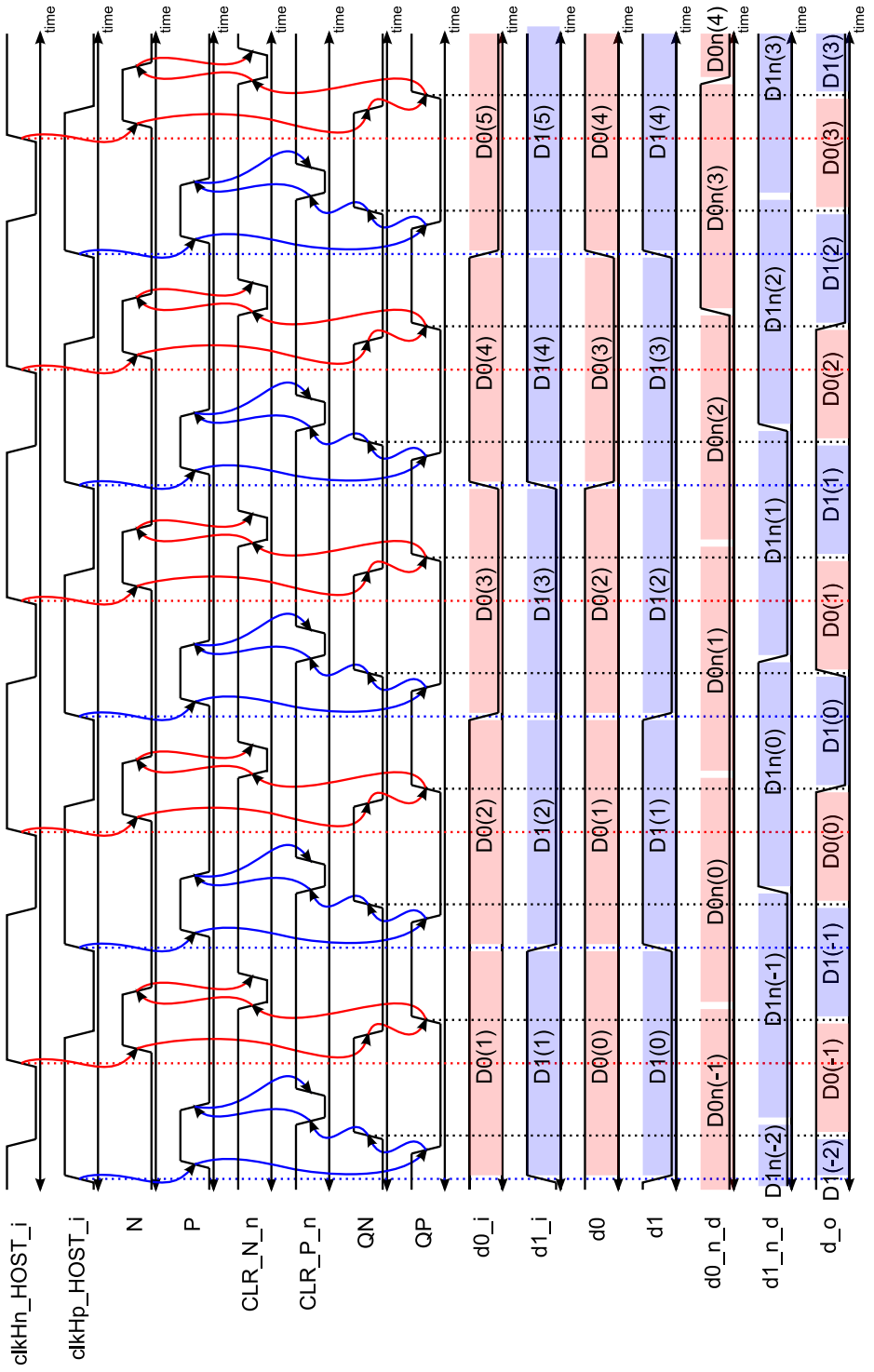


Figure 5.19: *SEN_DDR* cell timing diagram. Timing diagram for the *SEN_DDR* cell.

coming from the external memory. It can be observed in this timing diagram that every $2.4ns$ up to three packets coming from the 3D-DiRAM can be formed, in the case the three of them do not carry any read data information (128-bit packets). This could be the case of three write acknowledge packets. The legend *C1* and *C2* correspond to two 64-bit parts making the 128-bit command word seen in Table 5.2. In the case the packet carries read information, then additionally to the *C1* and *C2*, four more parts will be sent, the *D1*, *D2*, *D3* and *D4*. These four parts make up for 256 bits, for which the addition of the 128 *C1* and *C2* bits complete the 384-bit packet used, for instance, in the write or read answer packet. A 384-bit packet could be split into different $2.4ns$ clock cycles, as it can be seen for the case of the blue and gray packets. This division will force to keep in memory two $2.4ns$ clock cycles past samples for the deserialized bits, so that these split packets can be put together. Three packet ports are sent to the *Mux_Demux* block because of the possibility of up to three packets being received simultaneously in the deserialization process, as seen in Figure 5.20.

Due to the very disproportional dimensions in width and height of the *PADS_alignment* block (see Figure 5.1), specific architectures had to be designed for simple digital circuits such as multiplexers, demultiplexers, multiple input AND/OR/NAND-/NOR/XOR gates, etc. These architectures would be heavily pipelined if desired. When a signal needs to be demultiplexed into 64 different outputs, but those outputs are placed uniformly along over $13mm$, then high clock speeds would only be achieved

Function	Bits	Pos.	Description
Command segment			
Command	5	[4:0]	"00000" = NOP, "00001" = Read, "00010" = Write, "00101" = Successful read acknowledge, "00110" = Successful Write Acknowledge, "01001" = Failed Read Acknowledge, "01010" = Failed Write acknowledge
Packet size	2	[6:5]	Two are the possible packet sizes, "01" or "11". The packets carrying data such as write commands or answered read commands will be "11", for all the other packets "01".
Data or Command	1	[7]	Read commands or write acknowledges will have this bit at '0', the other commands will have '1'.
Framing	1	[8]	One of the bits in the framing sequence 0xF628.
Priority Flag	1	[9]	Not used.
Tag	14	[23:10]	Tag used to identify the answer to a command elevated in the 3D-DiRAM.
Destination node address	16	[39:24]	Not used.
Data	20	[59:40]	Part of the 256 bits of data in a packet.
Destination port address	4	[63:60]	Not used.
Return node address	16	[79:64]	Not used.
Memory Address	40	[119:80]	3D-DiRAM address from where to read or where to write.
ECC	8	[127:120]	Error correcting code.
Data segment			
Data	7	[6:0]	Part of the 256 bits of data in a packet.
Data or Command	1	[7]	Read commands or write acknowledges will have this bit at '0', the other commands will have '1'.
Framing	1	[8]	One of the bits in the framing sequence 0xF628.
Data	111	[119:9]	Part of the 256 bits of data in a packet.
ECC	8	[127:120]	Error correcting code.

Table 5.2: 3D-DiRAM packet format. A packet without data information such as a read command or a write acknowledge command will only be composed of a Command Segment. All the packets containing data information will have one Command Segment and two Data Segments.

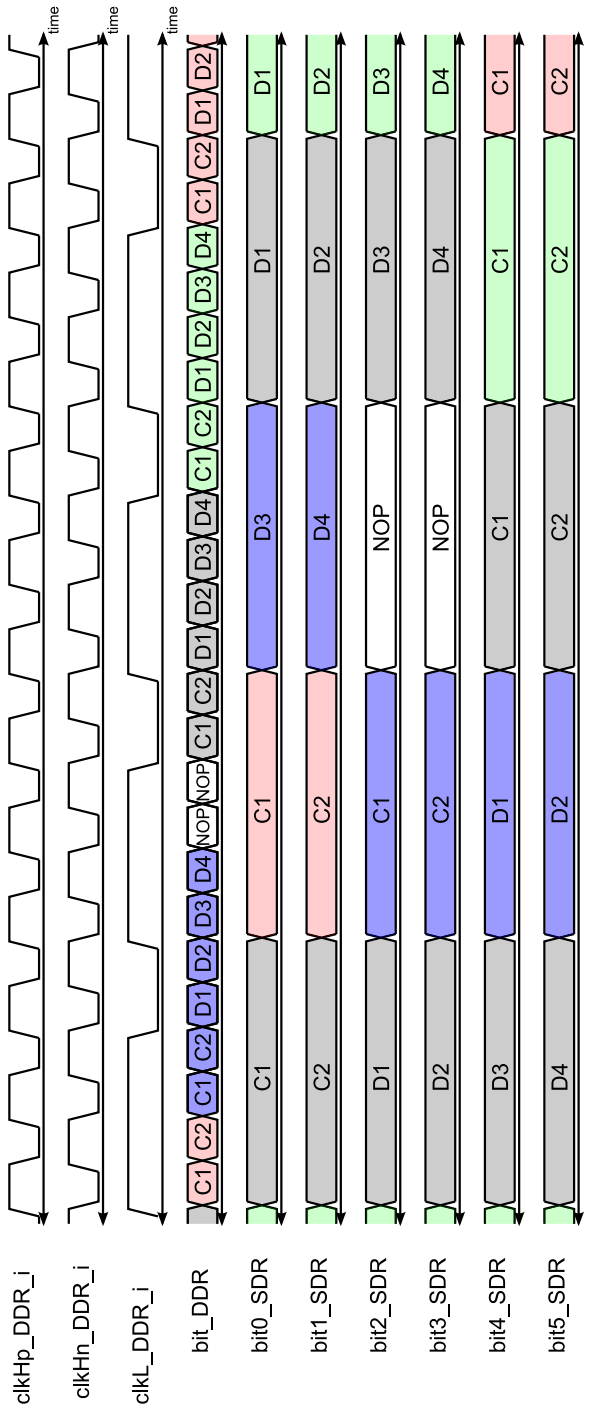


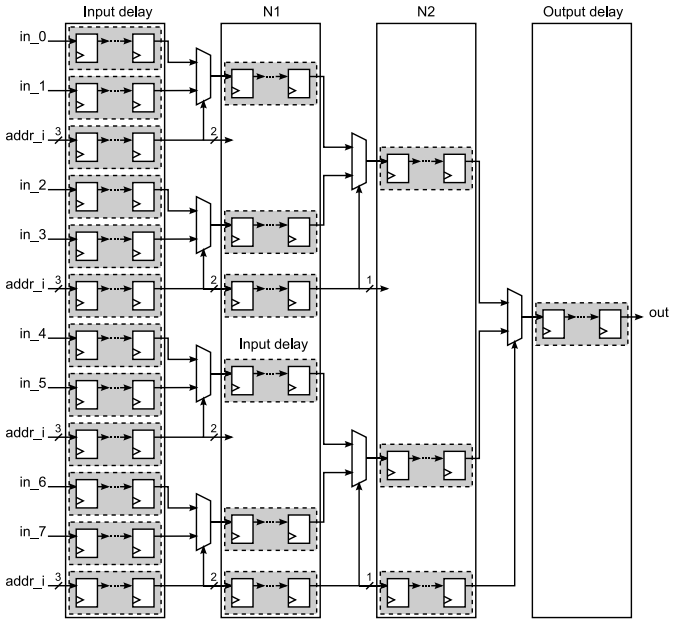
Figure 5.20: Deserialization of DDR bits. Timing diagram showing the deserialization of six DDR bits into six parallel bits using a three times slower clock frequency.

if serious pipelined is introduced. For this reason it is that in Figures 5.21 and 5.21 different pipelined architectures are presented. These architectures present parametric configuration for input/output delays and the number of register to use in all of the internal stages named NX, where X changes according to the position of the stage. Additionally, these architectures possess the option of removing register stages if the combinatorial logic in between stages is desired to be merged. For instance, if stage $N1$ for Figure 5.21b was removed, then the 2 input MUX at the input and output of that stage would be merged into a 4 input MUX. The constraint for these designs is that the number of input/outputs needs to be a power of 2. This would seem like a problem if the number of signals one desires to multiplex is not a power of 2, but when doing *logical synthesis*, if some of these inputs are left tied to '1' or '0', the corresponding registers will be automatically trimmed by the tool.

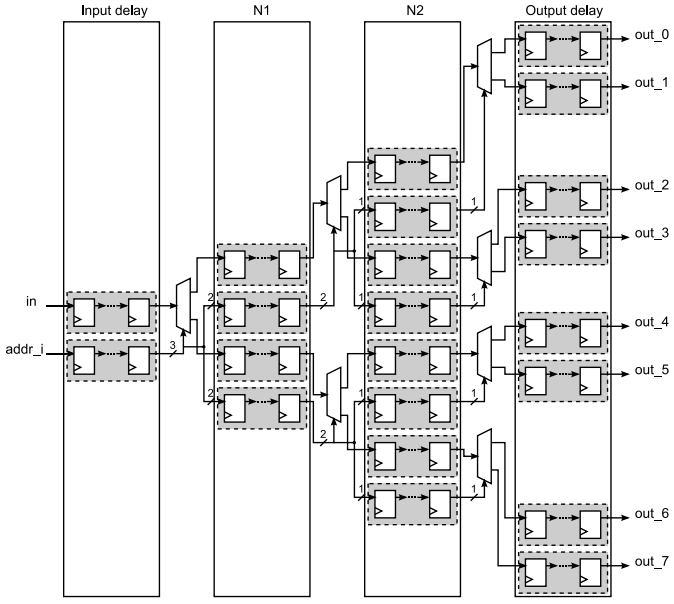
Figure 5.1 shows that the *PADS_alignment* block is not subdivided in any other blocks like the case of the *PAD_interface*. This makes it very difficult to take all of the output clocks coming from the two tree cells and just use them in the flow. *Place & Route* tools have a lot of problems in trying to use several clock inputs as different clock tree roots for the same clock signal. It is for this reason that a small program in *Matlab* was written to decide which register is clocked by which clock input. Very distinct steps can be found when performing *Place & Route*. The first three in order are *Floorplanning*, *Placement*, and *In place optimization*. After the third step the position of the registers does not suffer much change in subsequent optimizations,

and it is for this reason that upon finishing this step, all of the names of the registers, their clock source and positions are written to a file. This file is read by a *Matlab* program, and according to the position of the clock inputs and the position of the registers, all of the registers are reassigned a clock input according to their proximity to those clock pins. Once this assignment is finished, the netlist generated by the *Place & Route* tool is read by the program, and a reassignment of the clock sources for all of the registers is performed on that netlist file. After this process finishes, the resulting netlist is the one that needs to be used in a second *Place & Route* of the design. This double *Place & Route* approach had to be done not only for this *PADS_alignment* block, but also for the *Mux_Demux* and the *Network_1_interface* blocks.

Figure 5.22 presents the general architecture of the *PADS_alignment* block. All the blocks in blue are being clocked by the HOST clock and the ones in red by the DDR one. Because of the very long height of the block, many of the before mentioned pipelined architectures had to be used. For all of the pipelined trees, multiplexers, demultiplexers and the AND gate, either a result coming from all of the different *PAD-_interface* blocks had to converge to the physical middle point in the *PADS_alignment* block, or a value from this point had to be distributed to all of the *PAD_interface* blocks. For half of the height of the *PADS_alignment* block, 16 pipelining stages are required in order to achieve a $2.4ns$ clock period. All of the signals found on the bottom of Figure 5.22 will be used by the coordinating processor to set up the *DDR*

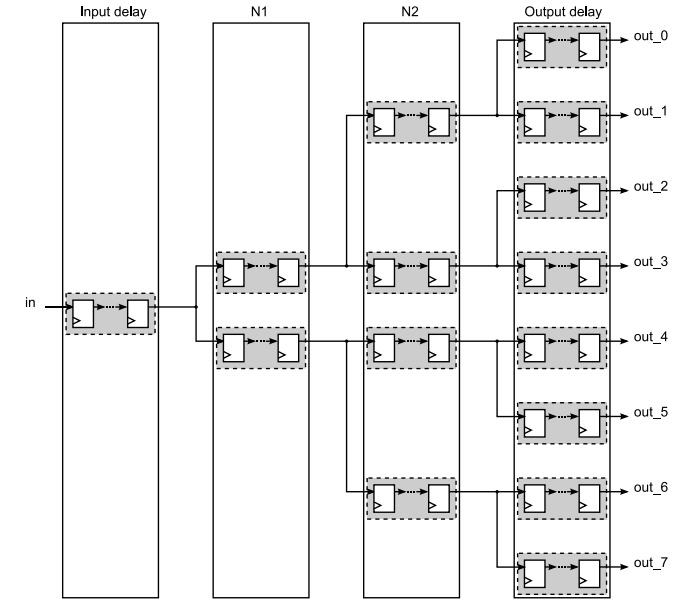


(a)

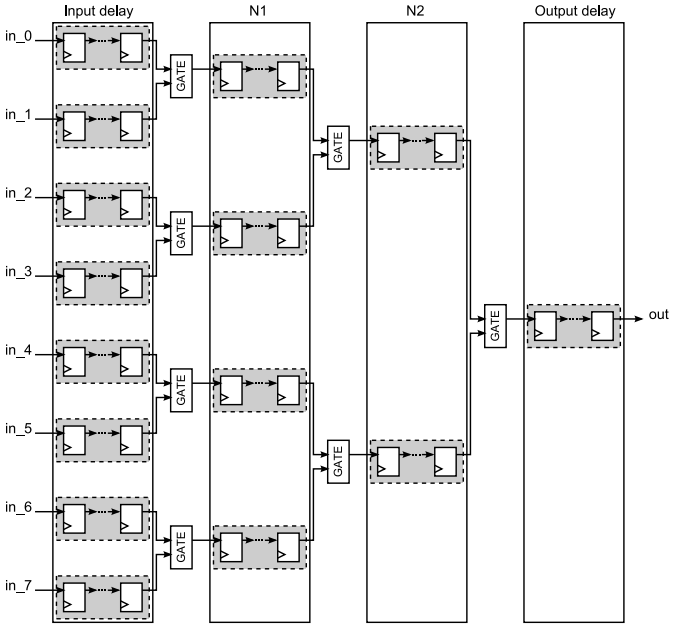


(b)

Figure 5.21: Pipelined Operations. Pipelined architectures used for when inputs or outputs for an operation such as AND/OR/NAND/NOR/XOR/MUX/DEMUX are spreaded over a very long distance. In Figure 5.21a the architecture for a pipelined multiplexer. In Figure 5.21b the architecture for a pipelined demultiplexer. In Figure 5.21c the architecture for a pipelined register tree. In Figure 5.21d the architecture for a pipelined gate such as AND/OR/NAND/NOR/XOR.



(c)



(d)

Figure 5.21: Pipelined Operations (cont.). Pipelined architectures used for when inputs or outputs for an operation such as AND/OR/NAND/NOR/XOR-/MUX/DEMUX are spreaded over a very long distance. In Figure 5.21a the architecture for a pipelined multiplexer. In Figure 5.21b the architecture for a pipelined demultiplexer. In Figure 5.21c the architecture for a pipelined register tree. In Figure 5.21d the architecture for a pipelined gate such as AND/OR/NAND/NOR/XOR.

DRAM PHY, but in order to take those output signals all the way to the bottom of the block, and input signals the other way, additional 16 stages shift registers were used.

The main function of the *PADS_alignment* block is set by the two *Parity Encoder and Decoder* block and the *Aligner* block. The *Parity Encoder and Decoder* block will receive the stream of bits from the 3D-DiRAM at $2.4ns$ clock speed, will parse the different packets shown in Figure 5.20 and check for errors using eight parity bits. If only one error was found, then this block will fix it, and will raise the output *one_error_HOST_o* high. If two or more errors were found, the output *two_error_HOST_o* will be asserted. Once these signals are asserted it is only through the input *reset_p_error_HOST_i* that the signals will be cleared. A sequence has to be followed with each packet received, set by the field "Framing" in Table 5.2. If that sequence is lost, the *Parity Encoder and Decoder* block will assert output *frame_error_HOST_o*. Again, only by asserting *reset_f_error_HOST_i* the *frame_error_HOST_o* signal is cleared.

In the flow coming from the *Mux_Demux* block, only one port will input a packet to be sent to the 3D-DiRAM. This packet is represented by the inputs *en_HOST_i*, *we_HOST_i*, *vector_HOST_i*, *addr_HOST_i* and *tag_HOST_i*. Every time *en_HOST_i* = *'1'*, *we_HOST_i* will determine if the packet is a read or write request. Signal *addr_HOST_i* will indicate from where or to where in memory the read or write command needs to take place. Input *vector_HOST_i* will carry the data in the case of

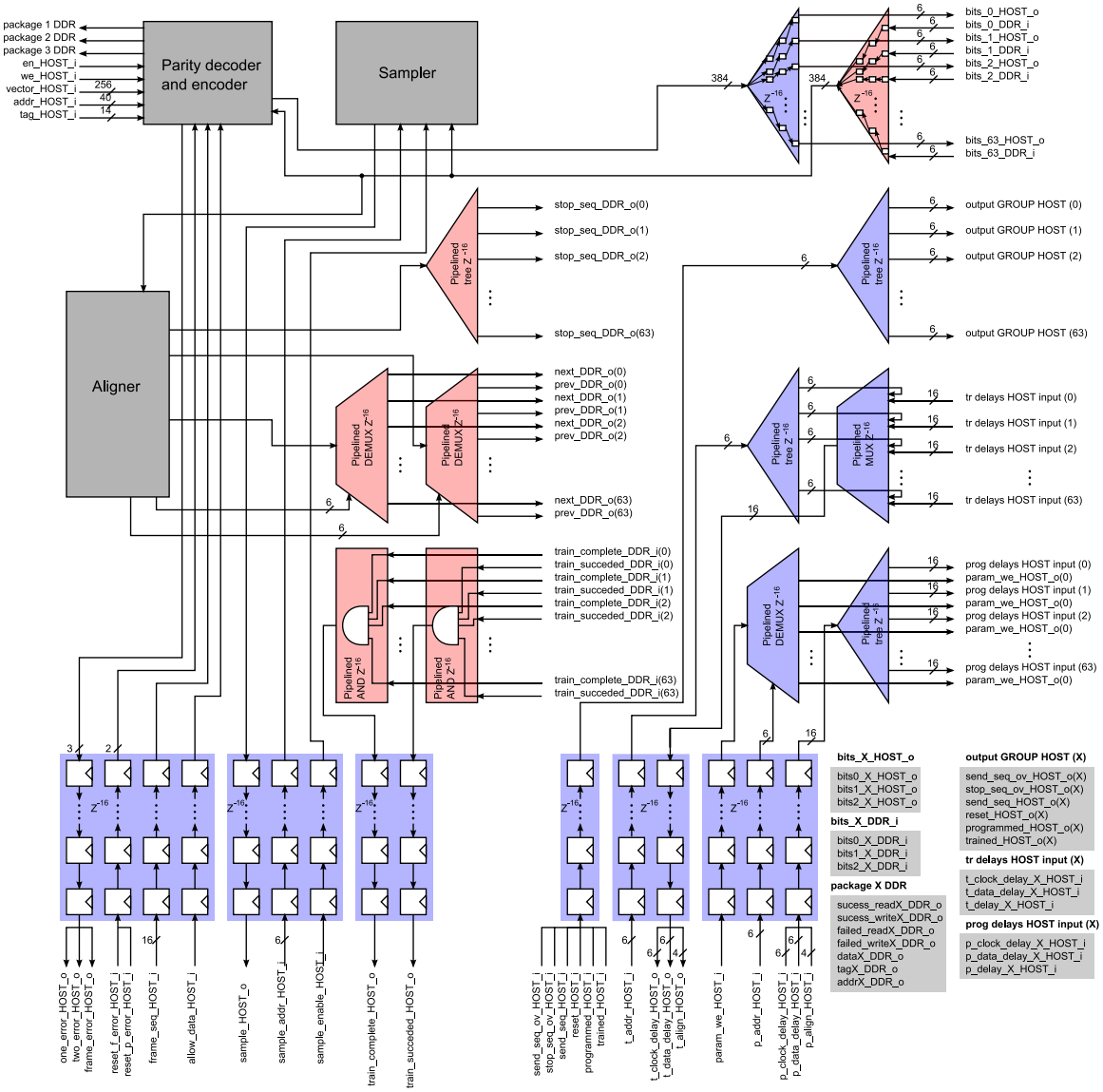


Figure 5.22: *PADS_alignment* block. General structure for the *PADS_alignment* block.

a write command, and input *tag_HOST_i* will be the tag associated to that particular packet.

For the case of the flow in the opposite direction, three packet output ports will be sent to the *Mux_Demux* because of the effect seen in Figure 5.20. Each of these ports has seven outputs. Output *sucess_readX_DDR_o* and *sucess_writeX_DDR_o* will indicate if the packet is a successful read command carrying the read data, or it is a write acknowledge indicating the correct write in memory. On the other hand, outputs *failed_readX_DDR_o* and *failed_writeX_DDR_o* will indicate an unsuccessful read or write command. Along with the before mentioned four outputs, *dataX_DDR_o*, *tagX_DDR_o* and *addrX_DDR_o*. These outputs are the data received in case a read command was elevated to the external memory, the tag expected to match the original read or write command sent to the external memory, and the address where that command performed a read or write in the 3D-DiRAM.

Just for debugging purposes, a *Sampler* block was added. This block would receive an address pointing to one of the 64-bit lines coming from the 3D-DiRAM, and using an enable signal, it would sample the received bits. The idea of this sampler is that, if any problem arises during the input pad training, one could read the values that are being read from every single line. The sample that is returned (*sample_HOST_o*) is a 16-bit signal because the expected training sequence coming from the 3D-DiRAM is a 16-bit word.

The last block needed to be addressed in Figure 5.22 is the *Aligner* block. This

block is the one responsible of performing training for the *second programmable delays*. Let's remember that for the case of the *second programmable delays*, the unit step for that delay is a $0.8ns$ clock cycle as seen in Figure 5.10. From 5.3.1, if $d_{prog} > P_{clk}$, then the maximum delay difference between any two bitlines coming from *PAD_interface* blocks can be considered $4.P_{clk}$ ($3.P_{clk}$ corresponding to the *interposer* routing mismatch, and one P_{clk} due to the *first programmable delay*). The *SEN_DELAY* cell in 5.3.3 achieves a minimum delay step of $65ps$, and then because 64 of these units are used for the *first programmable delay*, then a maximum $4160ps$ delay can be accomplished, satisfying $d_{prog} > P_{clk}$. In Figure 5.23 the algorithm used for performing the training on the *second programmable delays* is presented. The example presented here aligns eight streams, but in the case of the submitted chips, this training is done over the 64-bit lines coming from the 64 different *PAD_interface* blocks. The procedure implemented by the *Aligner* block for the example is the following:

1. Bit streams $2.i$ are compared with streams $2.i + 1$, for $i \in \{0, 1, 2, 3\}$. Four comparisons are shown at the same time in time t, $t + 1$, $t + 2$, $t + 3$, $t + 4$, $t+5$, $t+6$ and $t+7$. In reality these comparisons are done sequentially because there is only one sequence comparator. Let's consider the streams from *bits2* and *bits3*. The streams are not equal, and then a pulse is sent through the corresponding *next_DDR_o*, in this case *next_DDR_o(3)*. After the maximum shift of four, the sequences are still not matching, four pulses are then sent to *prev_DDR_o(3)* to revert to the original state of that signal. The other sequence

signal is the one that is now shifted. For the example shown *bits2* has to be now shifted to the right again three times for both *bits2* and *bits3* to match.

2. After the first set of comparisons, sequences $2.i$ and $2.i+1$ for $i \in \{0, 1, 2, 3\}$ will match. Consequently now the comparison is done between sequence $4.i$ and $4.i+2$ for $i \in \{0, 1\}$. Even if the comparison is done between two sequences, when deciding to shift one of them, the *prev_DDR_o* and/or *next_DDR_o* are pulsed also for the sequences that are known to be already in phase. For instance, in time $t + 9$, when *bit2* is shifted to the right, not only output *next_DDR_o(2)* is pulsed, but *next_DDR_o(3)* too.

3. Finally the last comparison is done between sequence $8.i$ and $8.i+4$ for $i \in \{0\}$.

For the explained example, the number of comparisons C is $8+4+2+1$, this can be expressed as:

$$C = \sum_{i=o}^{log_2(N\_streams)} 2^i \tag{5.7}$$

The number of comparisons done for the case of 64 3D-DiRAM bit lines is $32 + 16 + 8 + 4 + 2 + 1 = 63$.

5.4.2 Input/Output Signals

A brief description of what each of the input/output signals from block *PADS_alignment* are presented in Table 5.3.

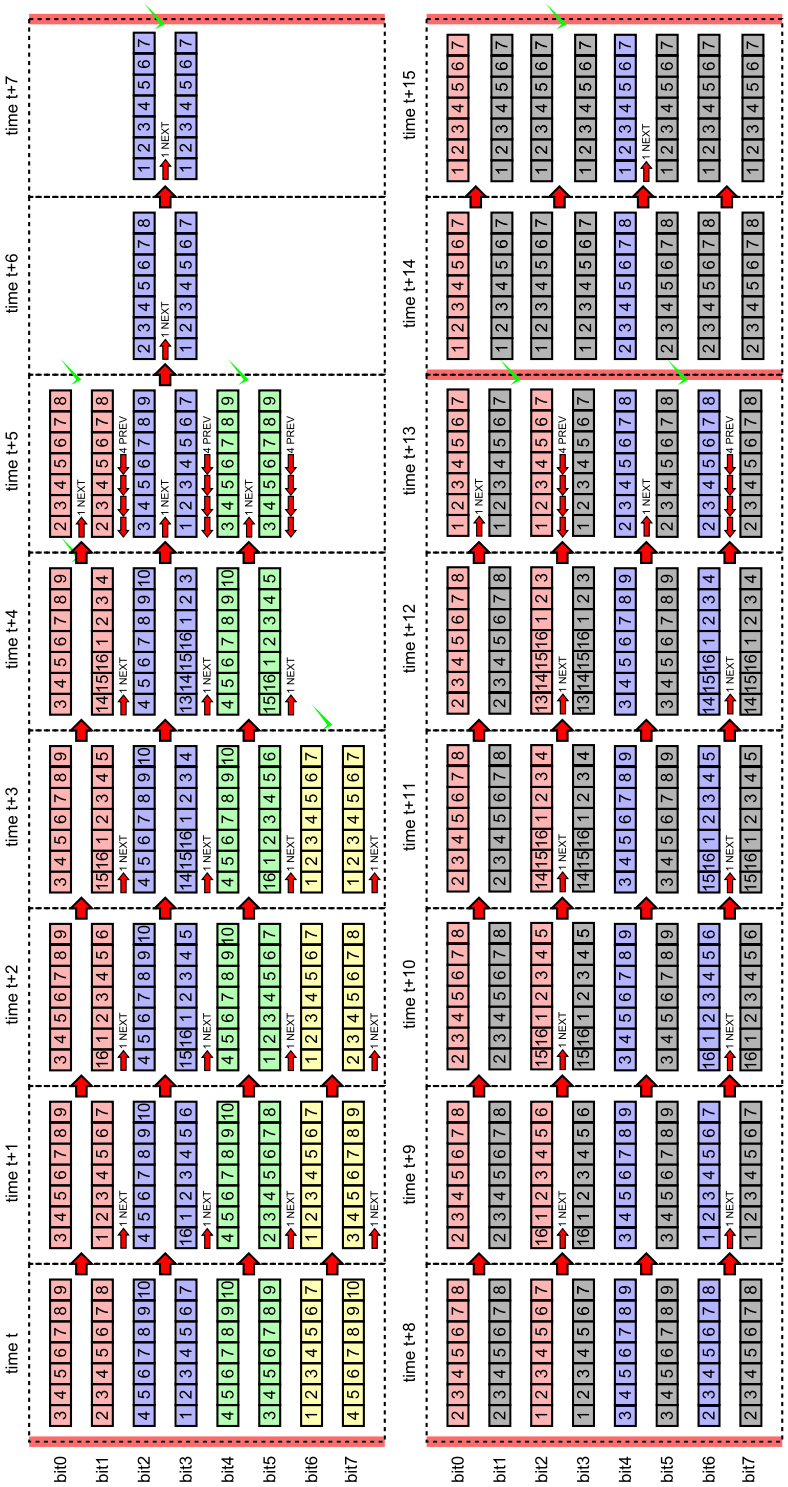


Figure 5.23: *Second programmable delay* training algorithm. This diagram presents step by step the way the *second programmable delay* is trained.

Table 5.3: Description of the *PADS_alignment* signals.

Signal name	Bits	O/I	Description
clkL_DDR_i	64	I	Clock for the path coming from the 3D-DiRAM. (2.4ns period)
clkL_HOST_i	64	I	Clock for the path going to the 3D-DiRAM. (2.4ns period)
Signals managed by the coordinating processor			
reset_HOST_i	1	I	Reset request. Upon power up, a reset pulse needs to be received.
send_seq_ov_HOST_i	1	I	Same as send_seq_ov_HOST_i from Table 5.1. In this case this signal is distributed to all of the *PAD_interface* blocks.
stop_seq_ov_HOST_i	1	I	Same as stop_seq_ov_HOST_i from Table 5.1. In this case this signal is distributed to all of the *PAD_interface* blocks.
send_seq_HOST_i	1	I	Same as send_seq_HOST_i from Table 5.1. In this case this signal is distributed to all of the *PAD_interface* blocks.
trained_HOST_i	1	I	Same as trained_HOST_i from Table 5.1. In this case this signal is distributed to all of the *PAD_interface* blocks.
programmed_HOST_i	1	I	Same as programmed_HOST_i from Table 5.1. In this case this signal is distributed to all of the *PAD_interface* blocks.
allow_data_HOST_i	1	I	When a reset pulse is sent to this block, all of the data signals sent to the *PAD_interface* blocks are zeros. It is only when a pulse is sent through *allow_data_HOST_i* that all of the packets received are forwarded.
train_complete_HOST_o	1	O	Indicator that the training process for the *first programmable delay* has completed. This signal is generated as the AND operation of all of the *train_complete_HOST_o* outputs from the *PAD_interface* blocks.
train_succeeded_HOST_o	1	O	Once *train_complete_HOST_o = '1'*, this signal indicates if the training process for the *first programmable delay* has been succesful or not. This signal is also generated by doing an AND operation among all the signals coming from all of the *PAD_interface* blocks.
align_complete_HOST_o	1	O	This signal indicates if the *second programmable delay* training has finished.
align_succeeded_HOST_o	1	O	Once *align_complete_HOST_o = '1'*, this signal indicates if the *second programmable delay* training process has been successful or not.
frame_seq_HOST_i	16	I	Frame sequence used for tagging each transaction sent to the 3D-DiRAM. The same sequence is expected for all of the packets coming back from the external memory. See *Framing* from Table 5.2.
t_addr_HOST_i	6	I	Signal *t_addr_HOST_i* addresses one of the *PAD_interface* blocks. It is through the outputs *t_clock_delay_HOST_o*, *t_data_delay_HOST_o* and *t_align_HOST_o* that the trained configuration delays are read back.
t_clock_delay_HOST_o	6	O	
t_data_delay_HOST_o	6	O	
t_align_HOST_o	4	O	

param_we_HOST_i	1	I	Input *p_addr_HOST_i* addresses one of the *PAD_interface* blocks. Using the input *param_we_HOST_i* as a write enable, inputs *p_clock_delay_HOST_i*, *p_data_delay_HOST_i* and *p_align_HOST_i* are used to write the manually configurable delay registers.
p_addr_HOST_i	6	I	
p_clock_delay_HOST_i	6	I	
p_data_delay_HOST_i	6	I	
p_align_HOST_i	4	I	
sample_en_HOST_i	1	I	Signal *sample_addr_HOST_i* addresses one of the 64 bit lines coming from the 3D-DiRAM. Signal *sample_en_HOST_i* will indicate when a 16-bit sample should be taken, and output *sample_HOST_o* will provide the sample.
sample_addr_HOST_i	6	I	
sample_HOST_o	16	O	
reset_p_error_HOST_i	1	I	Signals *one_error_HOST_o* and *two_error_HOST_o* will be asserted when one and two parity errors are found on the packets coming back from the 3D-DiRAM. It is only by asserting signal *reset_p_error_HOST_i* that the output signals are cleared.
one_error_HOST_o	1	O	
two_error_HOST_o	1	O	
frame_error_HOST_o	1	O	Signal *frame_error_HOST_o* will be asserted when the framing sequence received with each of the incoming packets is lost. Signal *reset_f_error_HOST_i* can be asserted for clearing the framing error signal.
reset_f_error_HOST_i	1	I	
Connections to the *Mux_Demux* block			
Path going to the *Mux_Demux* block			
en_HOST_i	1	I	Indicator of an incoming packet.
we_HOST_i	1	I	If the incoming packet corresponds to a write request, then *we_HOST_i* is asserted.
vector_HOST_i	256	I	Data field for an incoming write packet.
addr_HOST_i	40	I	Address for where to write or from where to read in external memory.
tag_HOST_i	14	I	Tag field for the incoming packet. The same tag is expected for the packet answering a read or write request.
Path coming to the *Mux_Demux* block. X can be 0, 1 or 2.			
Success_readX_DDR_o	1	O	Signal indicating a success read acknowledge packet.
Success_writeX_DDR_o	1	O	Signal indicating a success write acknowledge packet.
Failed_readX_DDR_o	1	O	Signal indicating a failed read acknowledge packet.
Failed_writeX_DDR_o	1	O	Signal indicating a failed write acknowledge packet.
dataX_DDR_o	256	O	Data field for the outgoing packet.
tagX_DDR_o	14	O	Tag field for the outgoing packet.
addrX_DDR_o	40	O	Address from where a read or write command was executed in the 3D-DiRAM.
Connections to the *PAD_interface* blocks			
Path going to the *PAD_interface* blocks. X can be 0, 1, 2, ..., 63.			
send_seq_ov_HOST_o	64	O	These outputs replicate input *send_seq_ov_HOST_i* with a delay of 32 clock cycles.

stop_seq_ov_HOST_o	64	O	These outputs replicate input *stop_seq_ov_HOST_i* with a delay of 32 clock cycles.
send_seq_HOST_o	64	O	These outputs replicate input *send_seq_HOST_i* with a delay of 32 clock cycles.
reset_HOST_o	64	O	These outputs replicate input *reset_HOST_i* with a delay of 32 clock cycles.
programmed_HOST_o	64	O	These outputs replicate input *programmed_HOST_i* with a delay of 32 clock cycles.
trained_HOST_o	64	O	These outputs replicate input *trained_HOST_i* with a delay of 32 clock cycles.
bits0_X_HOST_o	2	I	These three outputs make up for the six bits going in the path to the
bits1_X_HOST_o	2	I	3D-DiRAM that need to be converted into a single $0.8ns$ period DDR
bits2_X_HOST_o	2	I	signal.
t_clock_delay_X_HOST_i	6	I	Trained values for the *first and second programmable delays*. These
t_data_delay_X_HOST_i	6	I	values from the 64 different inputs will be multiplexed and provided
t_align_X_HOST_i	4	I	to the coordinating processor.
param_we_HOST_o	64	O	
p_clock_delay_X_HOST_o	6	O	Signals used for writing the manually chosen programmable delays in
p_data_delay_X_HOST_o	6	O	each of the *PAD_interface* blocks.
p_align_X_HOST_o	4	O	
Path coming from the *PAD_interface* blocks. X can be 0, 1, 2, ..., 63.			
train_complete_DDR_i	64	I	Signals indicating the completion of the *first programmable delay* training.
train_succeeded_DDR_i	64	I	Signals indicating if the completion of the *first programmable delay* training was successful or not.
bits0_X_DDR_i	2	I	These three inputs were originally six bits at DDR coming from the
bits1_X_DDR_i	2	I	3D-DiRAM. These six bits were deserialized into three two bit inputs
bits2_X_DDR_i	2	I	in the block *PAD_interface*.
stop_seq_DDR_o	64	O	These outputs command to the *PAD_interface* blocks to stop the transmission of the training sequence to the 3D-DiRAM.
next_DDR_o	64	O	These to signals are used in the training of the *second programmable*
prev_DDR_o	64	O	*delays*. Their usage can be seen in Figure 5.23.

5.5 The *Mux_Demux* Block

5.5.1 Operating Description

Figure 5.24 shows the block diagram for this unit. The *PADS_alignment* block receives only one packet transaction at a time in the flow from the CMP to the external memory, and then an organizing scheme had to be used to determine which of the eight packet ports from block *Port_interface* had to be forwarded to the *PADS_alignment* block. The best scheme found was the one of a token-ring. A token would circulate in the *Token RING* block in Figure 5.24, where each port would take the token, and if any packet is available to forward, then the correspondent *got_tokenX_HOST_i* input would be asserted. Because only one port can assert this signal at a time, the *AND filter* block in Figure 5.24 would output all zeros for the ports without the token. This allows the *Pipelined OR gate* block to perform the OR logical operation among all of the outputs from block *AND filter*. The output of the *Pipelined OR gate* would be whatever the port with the token has decided to forward. Once the port with the token has finished sending its packets, the *got_tokenX_HOST_i* is deasserted, and the token is forwarded to the next port. The token-ring proposed skips every other port for both vertical directions so that the traffic sent to the external memory could be more uniform.

With respect to the traffic on the opposite direction from the 3D-DiRAM to the CMP, one has to remember that three packet ports are provided by the *PADS-*

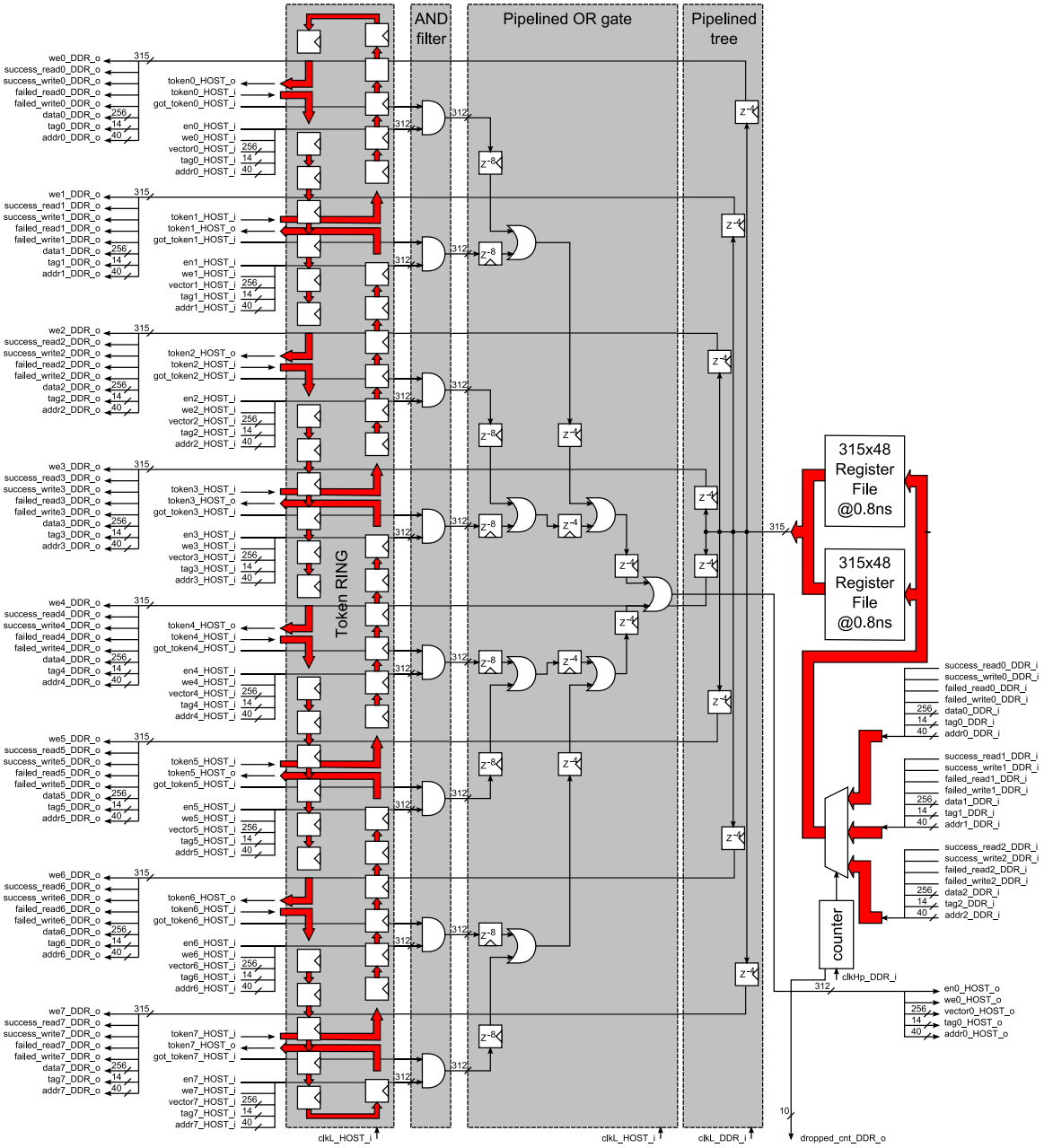


Figure 5.24: *Mux_Demux* block. General structure for the *Mux_Demux* block.

_alignment, because of the deserialization of bits done in the *PAD_interface* block. One needs to take into account that for every $2.4ns$ clock cycle going from the CMP to the 3D-DiRAM, only one packet can be transmitted. This packet is either a read or write command. On the way back from the 3D-DiRAM, the same average number of packets is expected, and then it can be assumed that each one of the three packet ports coming from block *PADS_alignment* can provide a maximum average of 1/3 of a packet. It is for this reason that, in order to be resilient to a change in that average packet traffic, a buffer had to be added. Two Register Files were added as seen in Figure 5.24, where two clocks are used, the *clkHp_DDR_i* and *clkL_DDR_i* ($0.8ns$ and $2.4ns$ period). Because a maximum of three packets can be received, then a multiplexer unit had to be implemented with a three times faster speed (*clkHp_DDR_i*). A ping pong architecture is then used for the two $315x48$ bits Register Files. On the other end of these register files, the clock used to extract the stored packets is the $2.4ns$ one (*clkL_DDR_i*). The output from these register files is then forwarded to all of the eight ports communicating with block *Port_interface*, and it is through the usage of three bits from the tag field that each port will decide if the incoming packet should be taken or not.

The block *Counter* in Figure 5.24 is the one responsible of controlling the multiplexing among all of the three input packet ports. Because of the absence of information of statistics on the traffic coming from the external memory, a reasonable guess had to be taken regarding the size of the register files. Every register file can

hold up to a maximum of 48 packets. If the average traffic from the external memory happens to be higher than expected (at least momentarily), packets could be lost. If this happens, output *dropped_cnt_DDR_o* will communicate the coordinating processor of this. If packets are lost, the whole processing flow the CMPs are supposed to perform could be broken, because the *Port_interface* blocks do not admit any packet loss. The idea, before starting the processing on-chip with the external memory, is to obtain some statistical information out of it. With this information, a few parameters controlling the packet traffic in the *Port_interface* blocks will be set so that no packets are lost.

Figure 5.25 shows the architecture of both register files used in this block. The physical synthesis had to be divided into tiles that together would build up the register file. Each of the tiles would allocate 45x48 bits. The reason for using hierarchy was that the clock used is a very high frequency one ($1.25GHz$), and flat *Place & Route* would not achieve that clock speed. The area used per bit is $17.8\mu m^2$, which is approximately double the size of the smallest register in the standard cell library. The area per bit is actually small considering the high operating frequency.

5.5.2 Input/Output Signals

A brief description of what each of the input/output signals from block *Mux_Demux* are presented in Table 5.4.

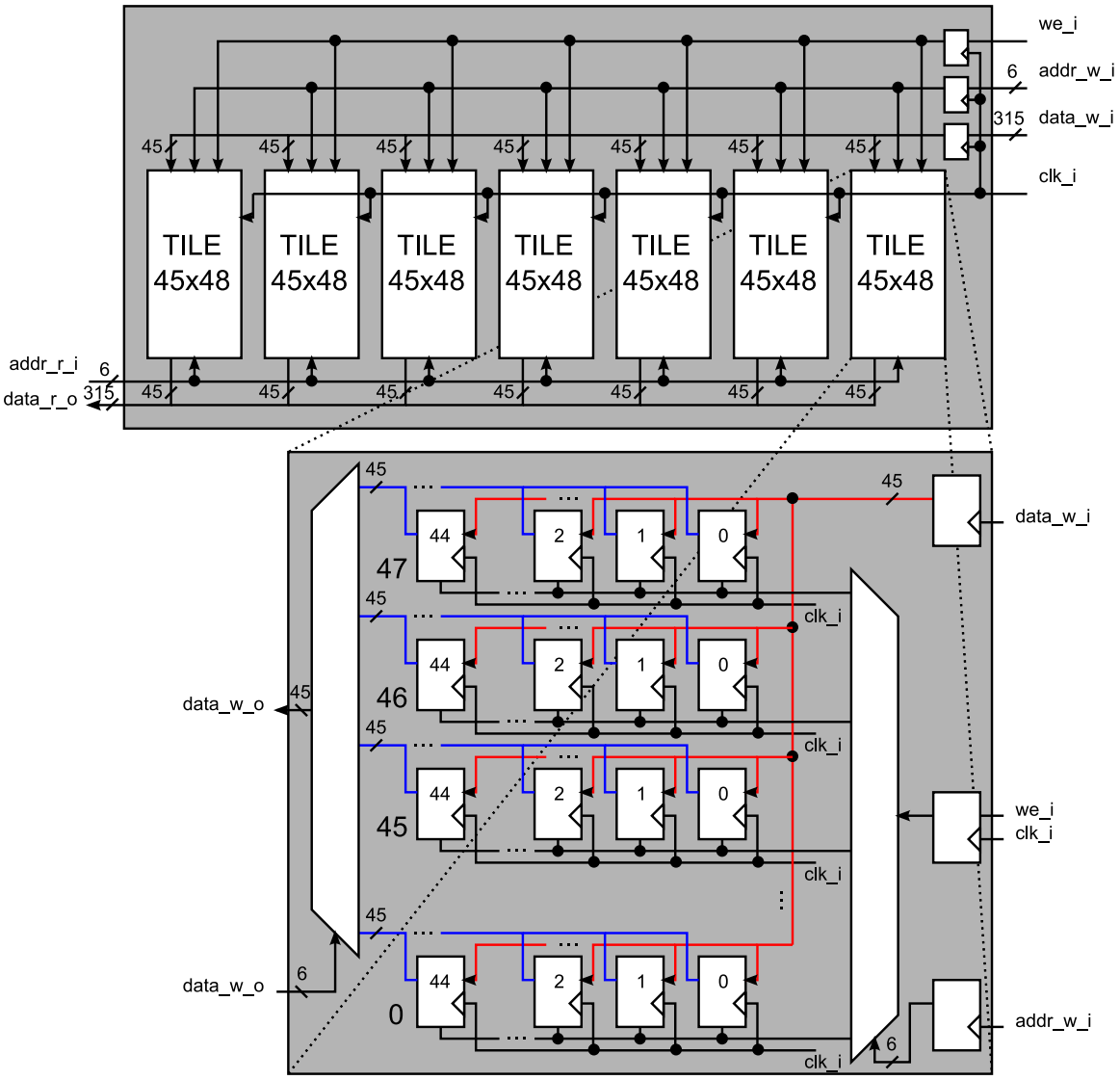


Figure 5.25: Register File architecture. Hierarchical design for the Register File used in block *Mux_Demux*.

Table 5.4: Description of the *Mux_Demux* signals.

Signal name	Bits	O/I	Description
clkL_HOST_i	64	I	Clock used for the flow from the CMP to the 3D-DiRAM. (2.4ns clock period)
clkL_DDR_i	64	I	Clock used for the flow from the 3D-DiRAM to the CMP. (2.4ns clock period)
clkH_DDR_i	64	I	High speed clock used for the flow from the 3D-DiRAM to the CMP. (0.8ns clock period)
Signals managed by the coordinating processor			
reset_i	1	I	Reset input.
dropped_cnt_DDR_o	10	O	Output signal identifying the number of packets lost in the path from the 3D-DiRAM.
Connections to the *PADS_alignment* block			
Path going to the *PADS_alignment* block.			
en_HOST_o	1	O	Indicator of an outgoing packet.
we_HOST_o	1	O	If the outgoing packet corresponds to a write request, then *we_HOST_o* is asserted.
vector_HOST_o	256	O	Data field for an outgoing write packet.
addr_HOST_o	40	O	Address for where to write or from where to read in external memory.
tag_HOST_o	14	O	Tag field for the outgoing packet. The same tag is expected for the packet returning in response from the 3D-DiRAM.
Path coming from the *PADS_alignment* block. X can be 0, 1 or 2.			
Success_readX_DDR_i	1	I	Signal indicating a success read acknowledge packet.
Success_writeX_DDR_i	1	I	Signal indicating a success write acknowledge packet.
Failed_readX_DDR_i	1	I	Signal indicating a failed read acknowledge packet.
Failed_writeX_DDR_i	1	I	Signal indicating a failed write acknowledge packet.
dataX_DDR_i	1	I	Data field for the incoming packet.
tagX_DDR_i	1	I	Tag field for the incoming packet.
addrX_DDR_i	1	I	Address from where a read or write command was executed in the 3D-DiRAM.
Connections to the *Port_interface* blocks. X can be 0, 1, 2, 3, 4, 5, 6 or 7.			
Path going to the *Port_interface* blocks.			
weX_DDR_o	1	O	Indicator of the existence of a packet.
Success_readX_DDR_o	1	O	Signal indicating a success read acknowledge packet.
Success_writeX_DDR_o	1	O	Signal indicating a success write acknowledge packet.
Failed_readX_DDR_o	1	O	Signal indicating a failed read acknowledge packet.
Failed_writeX_DDR_o	1	O	Signal indicating a failed write acknowledge packet.
dataX_DDR_o	256	O	Data field for the outgoing packet.
tagX_DDR_o	14	O	Tag field for the outgoing packet.

addrX_DDR_o	40	O	Address from where a read or write command was executed in the 3D-DiRAM.
Path coming from the *Port_interface* blocks.			
tokenX_HOST_i	1	I	Input through which a token is expected.
tokenX_HOST_o	1	O	Output through which the token is released after the corresponding port has finished forwarding packets.
got_tokenX_HOST_i	1	I	Signal that is asserted when a port has the token and desires to use it.
enX_HOST_i	1	I	Indicator of an incoming packet.
weX_HOST_i	1	I	If the incoming packet corresponds to a write request, then *weX_HOST_i = '1'*.
vectorX_HOST_i	256	I	Data field for an incoming write packet.
addrX_HOST_i	40	I	Address for where to write or from where to read in external memory.
tagX_HOST_i	14	I	Tag field for the incoming packet. The same tag is expected for the packet answering a read or write request.

5.6 The *Port_interface* Block

5.6.1 Operating Description

The *Port_interface* block introduced in Figure 5.1 is actually divided into eight single blocks. One can observe that the area enclosed in each of the eight blocks is the same. This was planned so that the rectangular shape of the *DDR_DRAM_PHY* could use all of the available silicon area. This approach would seem trivial, but it really requires much more work, as four *Place & Route* iterations need to be performed. *Logical synthesis* is done on a single design, and it is the resulting netlist the one used in the four *Place & Route* iterations. The result is then four designs with exactly the same functionality, but different physical shapes.

The main function of this block is the transmission of read and write transactions to the 3D-DiRAM from the *L1 network* through the *Network_1_interface* block, and vice-versa. Because transactions cannot be sent to the *Mux_Demux* at any time, a buffer was added on the path from the *L1 network* to the external memory. This buffer would accumulate memory transactions, and once this block receives the token from the *Mux_Demux* block, all of the transactions can be bursted out in the path to external memory. On the opposite direction, because packets cannot be inserted into the *L1 network* at any time, an additional buffer was added. The architecture for both registers used for both flow directions can be seen in Figure 5.26. The buffer in the CMP to 3D-DiRAM direction is named *Buffer NET to DDR*, and the one in the opposite direction *Buffer DDR to NET*. Both buffers are dual port register files, but they both contain particular characteristics that will be addressed when the step by step functioning of *Port_interface* block is analyzed.

For all of the previously addressed blocks in the *DDR_DRAM_PHY*, no voltage domains were crossed at any time. As a connection to the NoCs has now been presented, and considering that the NoCs have their own voltage domains, a passage from the NoCs' voltage domain to the *DDR_DRAM_PHY* voltage domain had to be addressed. Going back to Chapter 4, the communication between the PUs and the network node is an asynchronous four-phase handshaking protocol. This clock-less protocol allowed the level-shifting to take place without taking many considerations. This protocol could be used in the newly presented problem, but unfortunately this

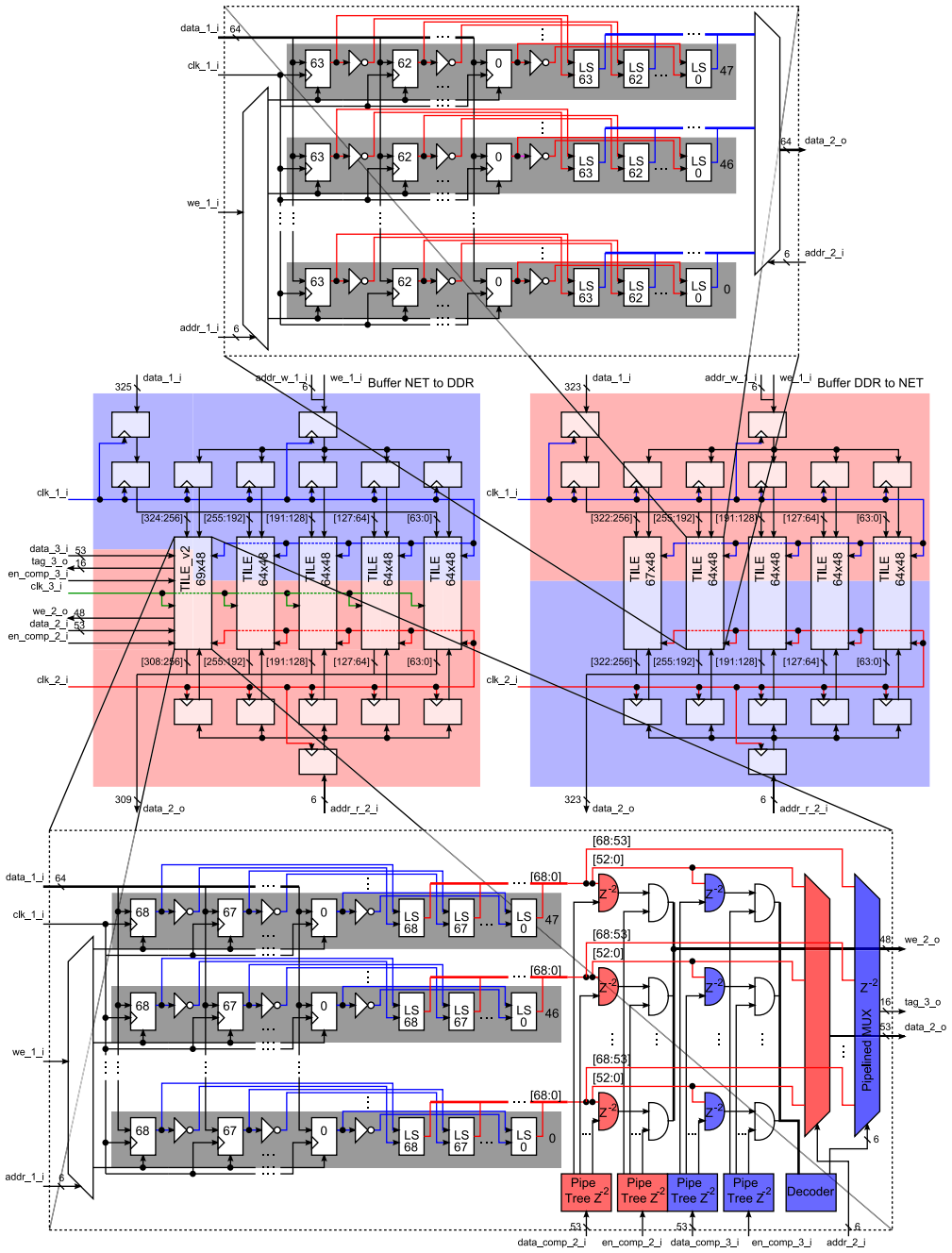


Figure 5.26: Buffers used in the communication between the *L1 network* rings through the *Network_1_interface* block and the *DDR_DRAM_PHY*. Architecture used for these buffers. The red and blue shaded regions represent the division of voltage domains. In red, blue and green the three different clocks used (network clock, clock in flow from the CMP to the 3D-DiRAM, and the clock for the data flowing in the opposite direction). In block *TILE_v2 69x48*, those blocks in red are clocked by *clk_2_i*, and the ones in blue by *clk_3_i*.

approach would result in a pretty slow interface to the *L1 network*. When the PUs of a row communicate to their *L1 network* ring network, their throughput suffers due to this clock-less communication approach, but because the same ring is shared among 8 or 16 PUs, the overall throughput does not suffer as much. On the other hand, if this protocol was used in the communication between the *DDR_DRAM_PHY* and the *L1 network*, this interface would have to deal with the traffic from all of the PUs in a *L1 network* ring using the same protocol each PU has in their communication to the *L1 network*. This would definitely be the bottleneck in the communication to the external memory.

The alternate approach to this protocol is the usage of the mentioned buffers. One of the *Buffer NET to DDR* ports could be provided to each of the *L1 network* rings. Several packets could be allocated in this buffer at the speed of the *L1 network*, and upon request, all of these packets could be sent to the 3D-DiRAM. But how is the voltage domain crossing problem solved? One would think that the buffer port used in the reading of the packets from *Buffer NET to DDR* could be just level-shifted to the *DDR_DRAM_PHY* voltage. This is a feasible approach only if the voltages used in the crossing are fixed and are never changed. As mentioned earlier, all of the voltages are tunable, and then when changing the two voltage domains, the delays suffered by the level-shifters would change as well. These changes would completely break all of the setup and hold analysis done by the *Place & Route* tool. The only solution found for this problem is the usage of a level-shifter cell for every single

storage element in the register files. Let's assume the *L1 network* writes register file positions from 0 to $N-1$. After this, a request to transmit the packets is elevated, and when the packets' transmission starts, packet in position 0 will be the first one to be sent. This means that the outputs of the level shifters for the first register file position, had all the time the *L1 network* took to write the N packets to settle its voltage values. This approach actually does not suffer throughput reduction due to voltage domain crossing at all. The only drawback to this approach is that the register files increase their size to double due to the presence of a level shifter for every single storage bit.

The chosen buffer size was 48. Then the size for both *Buffer DDR to NET* and *Buffer NET to DDR* was fixed to $323x48$ and $325x48$. This means that over 15000 bits have to be level shifted. For a regular single voltage domain register file this is not a problem at all, but when a voltage domain has to be crossed, and with it over 15000 signals, the situation becomes problematic. The metal stack used for this project has six lower capacitance metals, half of them are designated as vertical and half as horizontal. If one decided to use three of these metals (either the vertical or horizontal ones) to place the wires going from one voltage domain to the other, using a minimum pitch of $0.2\mu m$, $(15000/3)0.2\mu m = 1mm$ would be the total distance in the interface between voltage domains. This minimum pitch approach would never work due to routing problems, but assuming it will, and considering two register files per *Port_interface* block, over $16mm$ would be necessary in the interface between

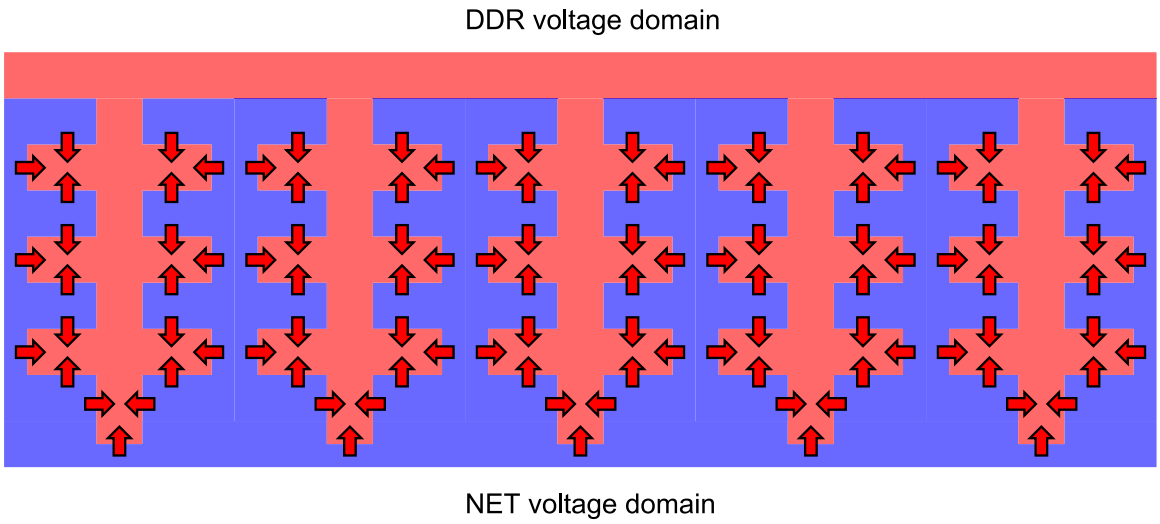


Figure 5.27: Register file voltage domain division. Both *Buffer NET to DDR* and *Buffer DDR to NET* were designed following this voltage domain division. The increase of contact surface between domains allowed for a level shifter to be added for every storage bit register.

voltage domains. This approach as it is, in a first approximation, would seem to be impossible to work. But, what if one had all of the required contact surface between the voltage domains? What if the line dividing the two voltage domains from the top of the *DDR_DRAM_PHY* to the bottom of it was not a straight line? Figure 5.27 shows the approach taken in designing these register files. It can now be seen that this approach increases significantly the space signals traveling from one domain to the other have. This is the approach that was taken to build both register files in the *Port_interface* blocks.

A pseudo architecture is presented in Figure 5.28 along with the steps taken in both paths from sending packets to the external memory, to the reception of their answers. One important characteristic of the architecture presented is that every

packet sent to the 3D-DiRAM is expected to generate an acknowledge packet for the packets writing in memory, and a read answer for the packets reading from memory. In table 5.2 the existence of a tag field allowed the unique recognition of transactions. When considering the big picture where up to 128 PUs will be injecting packets into the *L1 network*, 7 bits will be used to redirect the response packet to the PUs (3 for the eight token-rings, and 4 for the 16 different PUs in a row), leaving only 7 bits per PU to identify a response packet. This number of bits was considered too small for a general purpose distributed processing architecture. For this reason, an additional 16 bits are added to the tag field. This additional tag fields would not leave the CMPs, they stay local to the *Port_interface* block and are attached to the packet responses before sending them to the *L1 network*. The process by which transactions are elevated in the external memory is as follows:

1. In *Step1*, the *L1 network* uses one of the *Buffer NET to DDR* ports to write all of the desired transactions. After this, the number of transactions (that can be up to 48) will be passed through the level-shifted and synchronized signals *n_packet_NET_i* and *send_NET_i*. The first signal indicates the number of packets and the second one communicates to the block that the transmission can begin. The arrows in red correspond to the network clock domain (*clk_NET_i*). The arrows in blue correspond to the *clkL_HOST_i* clock in the path to the external memory. Once this block obtains the token from the *Mux_Demux* block, one by one the packets are taken their way to the 3D-DiRAM.

2. With *Step2*, after these packets were sent, response packets are expected. These packets now will be clocked by the *clkL_DDR_i* clock, represented by the green arrows. These packets will be also transformed into the *clkL_HOST_i* clock domain by the usage of a circular shift register. This shift register will have two pointers, where the read pointer is always ahead of the write pointer by one position. Both *clkL_DDR_i* and *clkL_HOST_i* run at the same speed, so, by just using a three stage circular shift register, it can be warrantied that the read pointer, which is governed by the *clkL_HOST_i* clock, will read a packet at least one clock period after it was written by the *clkL_DDR_i* clock, avoiding meta-stability problems. Two identical signals, clocked with two different clocks, carry the response packets to the *Buffer NET to DDR*. In Figure 5.26, *clk_2_i* for *TILE_v2 69x48* would correspond to *clkL_HOST_i*. In the flow to *Buffer NET to DDR*, part of the *clkL_HOST_i* clocked signal will be fed to the *data_comp_2_i* input in block *TILE_v2 69x48*. The bits fed to this input are the ones corresponding to the type of operation (2 bits), address (40 bits) and tag (11 bits). With the usage of *en_comp_2_i* as an enable signal, a comparison one to one is made of the *operation (2 bits) & address (40 bits) & tag (11 bits)* fields from the packet responses and the ones allocated in the *Buffer NET to DDR*. Additionally, as one can observe from block *TILE_v2 69x48* in Figure 5.26 the additional 16 bits tag added local to the *Port_interface* will be extracted from the packet that was successfully received. This extraction is done in the

clkL_DDR_i clock domain.

3. In *Step3*, the before mentioned signals will update the status of the *Correctly received packets* register, and the extracted extended tag field will be added to the received packet in the case of a successful read answer. The read answers will now be written in the *Buffer DDR to NET*. If, upon receiving acknowledges for all of the sent packets, some of the answers were flagged as failed, the process is iterated only for the failed transactions.

4. Once all of the answers were successfully received, and read answers written on the output buffer to the network along with the additional 16 tag bits, a output ready signal is asserted *ready_NET_o* to indicate the *N1 network* that the read responses can be read from the *Buffer DDR to NET*. The *L1 network* will be aware of the number of expected answers, so it will know how many positions from the *Buffer DDR to NET* need to be read.

5.6.2 Input/Output Signals

A brief description of what each of the input/output signals from block *Port_interface* are presented in Table 5.5.

Table 5.5: Description of the *Port_interface* signals.

Signal name	Bits	O/I	Description
port_n_i	3	I	Input identifying one of the eight different blocks. This input will be hardwired to the local port address.

reset_i	1	I	Reset input.
allow_data_i	1	I	After the *DDR_DRAM_PHY* has finished being configured, an assertion on this signal will allow the packet interchange between the *L1 network* and the *DDR_DRAM_PHY* block.
clkL_HOST_i	1	I	Clock used for the flow from the 3D-DiRAM to the CMP. ($2.4ns$ clock period)
clkL_DDR_i	1	I	Clock used for the flow coming from the 3D-DiRAM to the CMP. ($2.4ns$ clock period)
clk_NET_i	1	I	Clock used in the NoCs. ($300MHz$ clock)
Connection coming from the *Mux_Demux* block			
we_DDR_i	1	I	Indicator of the existence of a packet.
Success_read_DDR_i	1	I	Signal indicating a success read acknowledge packet.
Success_write_DDR_i	1	I	Signal indicating a success write acknowledge packet.
Failed_read_DDR_i	1	I	Signal indicating a failed read acknowledge packet.
Failed_write_DDR_i	1	I	Signal indicating a failed write acknowledge packet.
data_DDR_i	256	I	Data field for the incoming packet.
tag_DDR_i	14	I	Tag field for the incoming packet. (same tag as shown in Table 5.2)
addr_DDR_i	40	I	Address from where a read or write command was executed in the 3D-DiRAM.
Connection going to the *Mux_Demux* block			
token_HOST_i	I	1	Input through which a token is expected.
token_HOST_o	O	1	Output through which the token is released after forwarding packets.
got_token_HOST_o	O	1	Signal that is asserted when the block has the token and desires to use it.
en_HOST_o	O	1	Indicator of an outgoing packet.
we_HOST_o	O	1	If the outgoing packet corresponds to a write request, then *we_HOST_o = '1'*.
vector_HOST_o	O	256	Data field for an outgoing write packet.
addr_HOST_o	O	40	Address for where to write or from where to read in external memory.
tag_HOST_o	O	14	Tag field for the outgoing packet. (same tag as shown in Table 5.2)
Signals connecting the *L1 network* to the *DDR_DRAM_PHY*.			
Port in the flow from the *L1 network* to the *Port_interface*.			
we_NET_i addr_w_NET_i data_NET_i	1 6 325	I I I	Port used in the writing to the *Buffer NET to DDR*. Signal *we_NET_i* is the write enable input. Signal *addr_w_NET_i* is the address and *data_NET_i* the data to be written. The 325 bits correspond to *data (256 bits) & address (40 bits) & operation (2 bits) & tag (11 bits) & extended tag (16 bits).*
send_NET_i	1	I	Signal indicating the *Port_interface* block that the transmission can start.
n_packet_NET_i	6	I	Input indicating the number of packets to be sent to external memory.

Port in the flow from the *Port_interface* to the *L1 network*.			
ready_NET_o	1	O	Signal indicating the *L1 network* that all of the read answers are ready.
addr_r_NET_i	6	I	Address and data output for the *Buffer DDR to NET*. The *Network_1_interface* will use this signals to inject the response reads to the *L1 network*.
data_NET_o	1	O	

5.7 The *Network_1_interface* Block

5.7.1 Operating Description

The *Network_1_interface* is the block responsible of injecting packets into the *Port_interface* blocks, and reading the read answers back. The mechanism by which this is done is not trivial, as it is very easy to reach a state in which the *L1 network* rings are full of read and write commands and the *Port_interface* is not able to attend them. A local running counter will be used in the time slot assignment seen in 4.2. The buffer used in the flow to external memory in block *Port_interface* will be named *input buffer* and the one on the other direction, the *output buffer*. The *input buffer* will be considered to be *busy* when packets have been written in it and the command to forward them has been elevated. On the other hand the *output buffer* will be considered *busy* when the read answers have been collected and this buffer is trying to inject the answers into the *L1 network* rings. If a *L1 network* ring happens to be full of read and write requests, and both the *input* and *output buffers* on the *Port-*

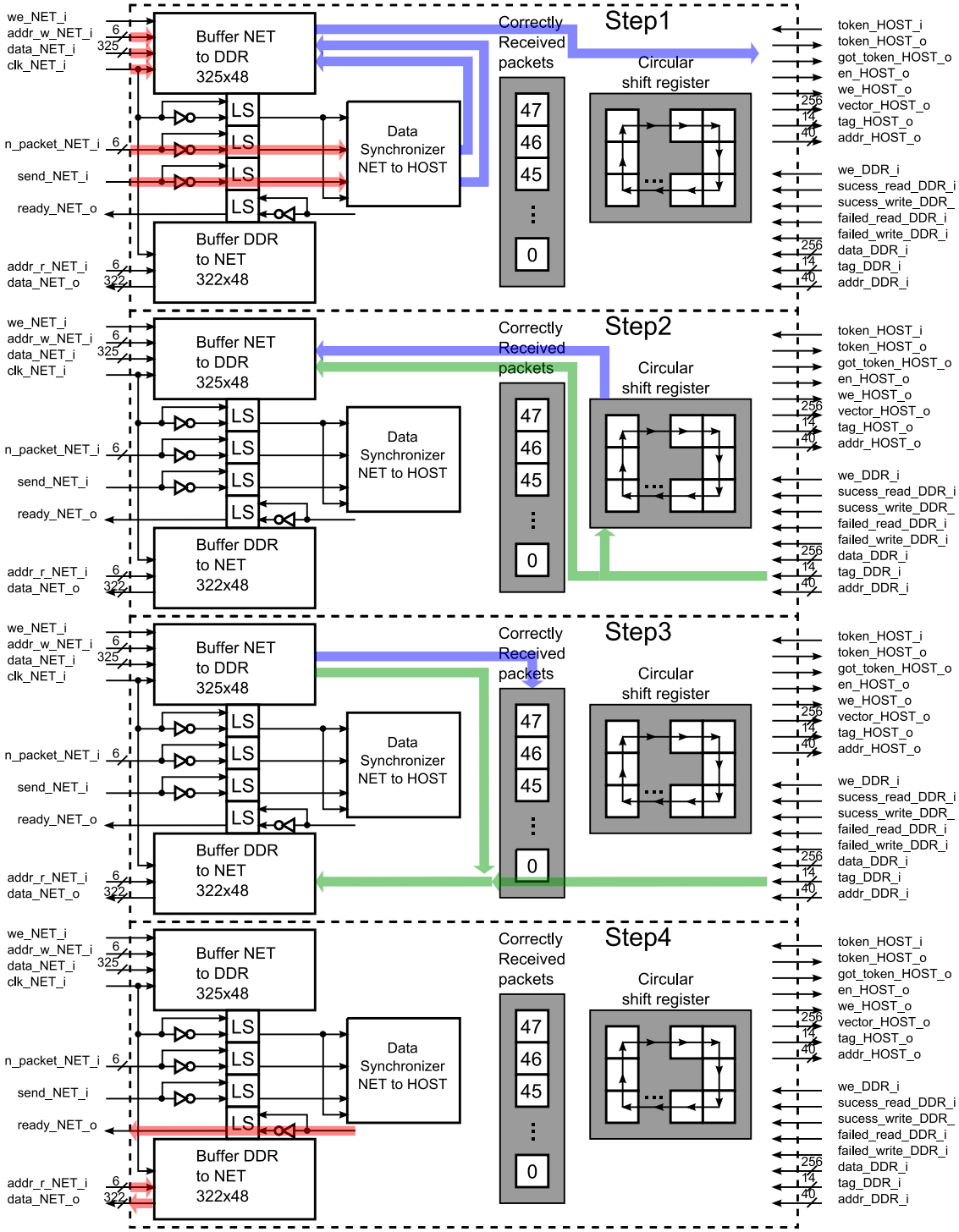


Figure 5.28: Step by step description of the functioning of the *Port-_interface* block. Green arrows represent flow of data clocked by *clkL_DDR_i*, blue arrows data clocked by *clkL_HOST_i*, and red arrows by the network clock.

_interface side are busy, then the *input buffer* is waiting for the *output buffer* to inject packets into the network and free the *output buffer*, to send the packet commands to the external memory. Unfortunately the *output buffer* cannot proceed because the network is full of read are write requests that will never go away unless the input buffer attends them. This is the kind of scenario that needs to be avoided.

It is clear now that both *input* and *output buffers* cannot be *busy* at the same time. Let's consider the following example. After power-up, both *input* and *output buffers* will be empty. The *input buffer* will start collecting packets and eventually these packets will be sent to the external memory. While these packets are sent to the 3D-DiRAM, the *input buffer* will be flagged as *busy*. When finally all of the packets are sent, and if any of those packets sent were read requests, then the *input buffer* will get free and the *output buffer* will get busy trying to inject the read responses into the token-ring network. If none of the packets sent to the 3D-DiRAM are read commands, then no read response is expected, then the *output buffer* will not get busy, and the *input buffer* will be able to immediately start collecting new packets to forward. If on the other hand, the *output buffer* is busy trying to inject packets into the network, the solution to the dead-lock found is that the *input buffer* will only accept packets from the network if at the same time a packet from the *output buffer* is injected into the network. This makes sure that the scenario where the network is full of read and write commands is not a dead-lock problem.

An additional feature that was incorporated, is a programmable time counter

that senses when packets have not been written into the *input buffer* for a while. There is no need to completely fill the *input buffer* to forward the packets to the 3D-DiRAM. If the buffer is not completely filled, the traffic going to the 3D-DiRAM will decrease, allowing to mitigate problems related to packet dropping seen in 5.5. It is not unlikely that at some point one of the *L1 network* rings would become empty. If they do and the *input buffer* hasn't been filled, the absence of the time counter would make the transmission of the already written packets in the *input buffer* halt for an undetermined amount of time. The existence of this time counter fixes this, elevating a transmission request after a time of absence of activity in the *input buffer*. A caveat to take into account is that this time counter would only work if the *output buffer* is not busy, otherwise it is risked to have both *input* and *output buffers* busy at the same time.

Figure 5.29 presents the step by step explanation of the interchange of packets from the *L1 network* ring and the *Port_interface*. The steps are the following:

1. After power-up, the *Network_1_interface* will start generating empty packets with the slot addresses that will allow the different PUs to insert read or write commands into the *L1 network* ring. The time for this is shown as $time = 0$ in Figure 5.29.

2. At time $t = A, A >> 1$, many PUs will have inserted packets into the ring network and some of them will have been taken by the *input buffer*. Upon the write of these packets into the *input buffer*, a *NOP* packets are inserted back

into the network with a slot number generated by a local running counter.

3. After the before mentioned time counter expires or the *input buffer* is fully filled, the transmission of packets to external memory is started (this is seen in $time = A + B, B >> 1$ in Figure 5.29). After a while answer packets will start flowing from the external memory into the *output buffer*. In this case two read answers were recorded.

4. At time $t = A + B + 1$, a NOP packet was received by the *Network_1_interface* block, and because it is an empty packet, it is replaced back to the ring network by one of the read answers. The slot number for this answer will not follow the expected pattern, as it could belong to any of the PUs. This decision was taken in order not to slow down the injection of read answers into the ring network.

5. At time $t = A + B + 2$ a similar situation arises, but the packet taken by the *Network_1_interface* block is a read packet, and then upon writing this packet in the *input buffer*, the *output buffer* is allowed to inject one of his answer packets into the ring network.

5.7.2 Input/Output Signals

A brief description of what each of the input/output signals from block *Network_1_interface* are presented in Table 5.6.

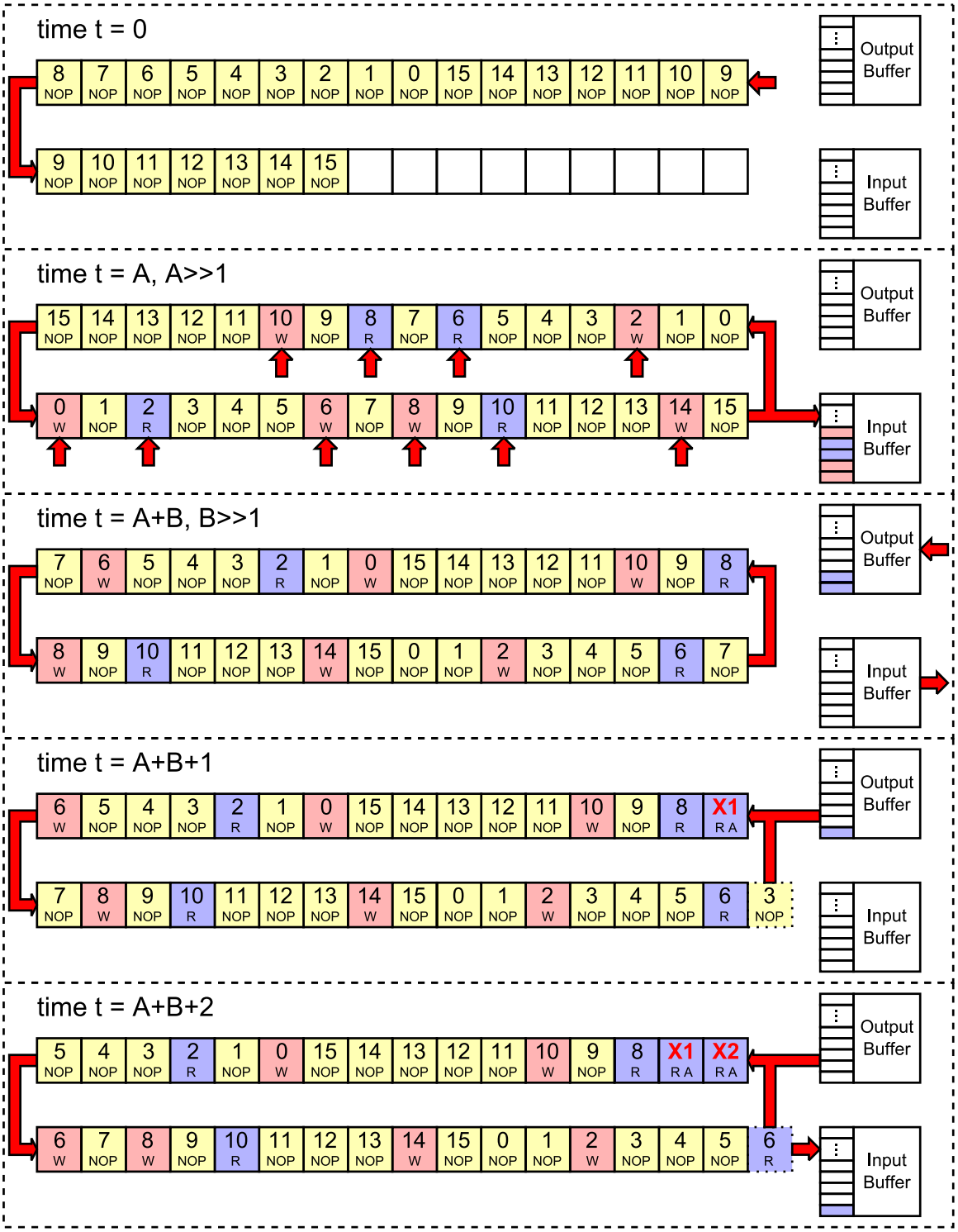


Figure 5.29: Step by step explanation of the interface between the *L1 network* ring and the *Port_interface*.

Table 5.6: Description of the *Network_1_interface* signals.

Signal name	Bits	O/I	Description
clk_X_i	1	I	Clock input for each of the eight different *Network_1_interface* blocks. X can be 0, 1, 2, ..., 7.
reset_n_X_i	1	I	Reset input for each of the eight different *Network_1_interface* blocks. X can be 0, 1, 2, ..., 7.
network_empty_X_o	1	O	Output indicating the absence of transactions in both *input* and *output buffers*. This signal will be provided to the coordinating processor, to coordinate processing phases.
Signals managed by the coordinating processor.			
max_words_i	6	I	Input used to constraint the maximum number of packets that can be written to the *input buffer*. This number has to be less or equal to 48.
max_time_counter_i	10	I	Number of clock cycles of inactivity at the *input buffer* that triggers the transmission of the written packets in this buffer to external memory.
Interface connecting to the *L1 network*. X can be 0, 1, 2, ..., 7.			
tag_addr_LR_X_i	16	I	See Table 4.5.
addr_LR_X_i	40	I	See Table 4.5.
tag_LR_X_i	11	I	See Table 4.5.
op_LR_X_i	2	I	See Table 4.5.
data_LR_X_i	256	I	See Table 4.5.
tag_addr_RL_X_o	16	O	See Table 4.5.
addr_RL_X_o	40	O	See Table 4.5.
tag_RL_X_o	11	O	See Table 4.5.
op_RL_X_o	2	O	See Table 4.5.
data_RL_X_o	256	O	See Table 4.5.
Interface connecting to each of the *Port_interface* blocks. X can be 0, 1, 2, ..., 7.			
Port connecting to the *input buffer*.			
send_X_o	1		See Table 5.5.
n_packet_X_o	6		See Table 5.5.
we_X_o	1		See Table 5.5.
addr_w_X_o	6		See Table 5.5.
data_X_o	325		See Table 5.5.
Port connecting to the *output buffer*.			
addr_r_X_o	6		See Table 5.5.
data_X_i	322		See Table 5.5.
ready_X_i	1		See Table 5.5.

Chapter 6

Subthreshold CMOS Library Design

6.1 Introduction to Synthesis

Two flows can be found in the translation from *VHDL/Verilog* description code into physical layout. The first flow, usually called *Logical Synthesis*, corresponds to a logical translation of an RTL description into a netlist, using only cells corresponding to a particular standard cell library. The second flow, usually called *Place & Route*, uses as input the resulting netlist, it takes care of laying down all of the physical cells in that netlist, and it additionally performs the required metal interconnections among the cells.

In starting the first flow, a standard cell library is required. This library will

contain mainly memory elements (such as registers and latches), and logical elements allowing to translate any truth table into a gate level netlist (such as AND, OR, XOR, etc., gates). Due to the fact that area and speed are usually constraints in a design, several versions for a given cell can be available in a library. For instance, if a 4-input AND gate was required, then that logical function can be designed by using three 2-input AND gates, or maybe just one cell is required, if that 4-input AND gate is already part of the library. With the second choice, it is very likely that the silicon area used will be smaller, otherwise that 4-input AND gate would not be part of the library. Some other times, if speed is more important than the reduction in area, the *logical synthesis* tool might find more convenient to use three 2-input AND cells instead of a single 4-input AND cell. Furthermore, the tool might find convenient to change the strength of each of those 2-input AND gates. A standard cell library might have several versions for the same logical function, and that is because the driving strength of these cells can be different.

Two types of files mainly characterize a standard cell library, the first one is the timing library (*.lib* extension file), and the other one characterizes the geometry of the cells geometrically (*.lef* extension file). The first file characterizes the rising and falling transitions at the output of each of the cells, as well as the input-to-output delay. This is accomplished by simulating the schematic of each of the cells, changing both input slew rates and output capacitances. The results are bi-dimensional matrices characterizing not only output falling and rising transitions, and input-to-

output delays, but also power dissipation. In both *Logical Synthesis* and *Place & Route* tools, these tables are interpolated to achieve more accurate timing results. If MMMC (Multi-Mode, Multi-Corner) is used, multiple corners are considered in the synthesis process. Both *nfet* and *pfet* transistors can be characterized by current-voltage curves, but these curves are not unique. Due to fabrication variations, each of these curves correspond to a sample from the statistical distribution of curves characterizing a device. One can then extract timing matrices characterizing standard cells for different statistical cases. Usually four are the cases provided by a foundry, fast-fast, slow-slow, fast-slow and slow-fast for the cases of both *nfets* and *pfets*. Due to environmental conditions as well as power dissipation conditions, temperature is another variable considered. One can then generate several *.lib* files where temperature, voltage supply and *nfet-pfet* corners can be changed. All of these conditions are taken into account by both *Logical Synthesis* and *Place & Route* tools to make sure that timing constraints are satisfied under any condition.

After *Place & Route*, a resulting *.gds* file is generated, but in order to obtain it, the layout of each of the standard cells is required. Some of the information provided by the layout of a standard cell is not really necessary in performing *Place & Route* of a design. One, for instance, does not need to know the characteristics of each of the transistors in a standard cell, that has been already characterized by the timing in a *.lib* file. *Place & Route* only make use of metals to perform connections, and then only the metal information of a cell is required. It is for this reason that a

less informative version of the layout of a cell is used. This version will only contain information of the position of the metal pins, and sizes for each of the cells in the library. It is at the end of the *Place & Route* flow that these *.lef* cells will be replaced by the real *.gds* versions.

Figure 6.1 presents a diagram showing the different synthesis steps when doing both *Logical Synthesis* and *Place & Route*. The minimum requirements when doing *Logical Synthesis* are the standard cell library timing information, all of the design constraints through a *.sdc* file, and the design to synthesize either in *VHDL*, *Verilog* or both. The *Logical Synthesis* does not usually take into account capacitance or resistance of wires in performing the interconnection among cells, it usually only considers the timing information provided by the *.lib* timing library files. The result of this step is a *Verilog* gate level netlist, with an updated constraints file. If a better approach, with more accurate timing constraints is desired, one can opt to perform a *Logical Synthesis* in topological mode. With this approach one can provide minimum information about capacitances and resistances of wires through an additional *.lib* file containing basic information about the metal stack. With this file, the *Logical Synthesis* tool can make a first order estimation of delays between cells. If one already has information about the floorplan of the design being sythesized, one can additionally provide that information to the tool through a *.def* file. This file will have basic information about the floorplan of the design such as pin location and floorplan sizes. The resulting gate level netlist can be logically verified by simulating it in tools such

as *Modelsim*.

The *Place & Route* step will take the resulting gate level netlist and will perform the physical placement and interconnection of the standard cells. One can usually use the same constraints file used for *Logical Synthesis*, but sometimes, because of name remapping, it is recommended to use the file generated by the *Logical Synthesis* tool. Additional to the standard cells timing information, the constraints file, and the gate level netlist, a few more files are necessary in performing *Place & Route*. Because layout will be generated with this step, one needs the geometrical information about the cells, which is provided through the *.lef* cells. An additional *.lef* file will be provided, the one characterizing the metal stack. This file will provide the tool information about the metal stack such as minimum DRC rules to follow when routing, and information of how to create vias for the connection of two wires in different metals. For resistance and capacitance parasitics, the original *.lib* file for *Logical Synthesis* will now be replaced by more accurate descriptive files. These files can be either captables (*.captbl* extension file) or QRC (*.qrc* extension file) files. QRC files are usually used for feature sizes under $80nm$, and captables are more common in bigger feature sizes. By replacing the *.lef* versions of the standard cells by real *.gds* layouts, a layout of the synthesized design is generated, along with a gate level netlist. This netlist can be used again in tools such as *Modelsim* to verify the correct functioning of the netlist. One can additionally perform static time analysis, which is done by additionally incorporating the delays suffered by each of the cells in the

design (*.sdf* extension files) to the simulation in *Modelsim*. Just for safe measures, by importing the gate level netlist to tools such as *Cadence Virtuoso*, DRC and LVS can be performed on the resulting layout to further verify the correctness of the resulting *.gds*.

6.2 Standard Cell Library Design

When aiming for power reduction, two main aspects are considered in CMOS circuitry, leakage currents and switching activity. The reduction of power consumption by lowering switching activity is one possible approach, where one can drive the design of architectures by estimating switching activity as seen in,[34–36] or maybe come up with new encoding mechanisms that warranty this reduction like in.[36] The approach taken for this project was the reduction of the voltage supply in order to reduce power consumption. The energy in a capacitor is well known to be $\frac{1}{2}CV^2$, and then for CMOS, the power reduction scales down quadratically with the supplied voltage. With this approach leakage currents can be significantly reduced, especially when considering subthreshold voltages.

The design of a standard cell library is not a trivial matter, a lot of time needs to be put in the design of every single cell so that an efficient design can be accomplished. In the design of these cells several unconventional CMOS architectures were analyzed, like the Schmidt-Trigger approach from[37] which allows the voltage supply

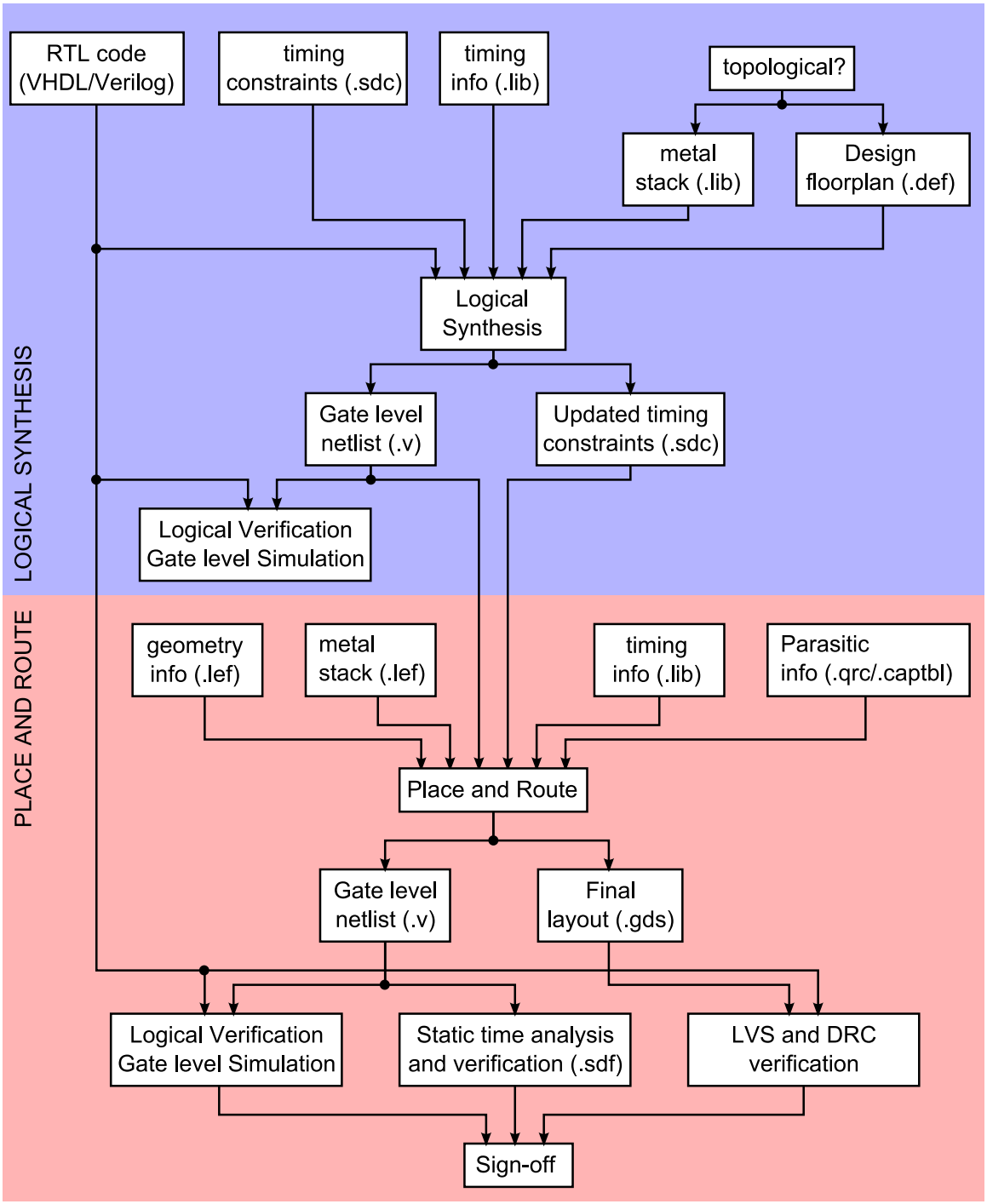


Figure 6.1: Synthesis flow diagram. Diagram showing both *Logical Synthesis* and *Place & Route* steps in the synthesis of a design.

to go below $100mV$. Such a reduction in the power supply decreases dramatically power dissipation, at expense of a dramatic reduction in the frequency of operation and a significant increase in the size of the standard cells. This approach was simulated, but was discarded as it would have reduced number of PUs to less than half. Additionally, the hysteresis inherent to a Schmidt-trigger cell made it very difficult to perform timing characterization on these cells. Simple combinatorial cells such as a AND or OR gates would not be considered combinatorial any more in this architecture. Memory is inherent to each of these cells, and then automation in the timing characterization of these cells using tools such as *Cadence Liberate* would become a very complicated task. Furthermore, *logical synthesis* tools such as *Synopsys DC Compiler* or *Place & Route* tools such as *Cadence Innovus*, are not prepared to deal with this kind of memory.

As a result, the design of a standard cell library was decided to be started based on an already tested standard cell library provided by *IBM*. The nominal voltage for the provided library is $1.2V$, and then if subthreshold voltages were decided to be used, then modifications to these cells would have to take place.

As a first step all cells containing more than two transistors in the path from the output to the voltage supply for the *pfet* transistors, or in the path from the output to ground for the case on *nfet* transistors, were discarded. Secondly, considering a power supply of $300mV$, all of the remaining cells were simulated, utilizing the same output capacitance and same input slew. All of those cells whose input to output

delay deviated considerably with respect to the standard inverter, were discarded as well. Many cells such as clock gaters, registers with asynchronous reset or preset, and almost all of the cells containing more than two inputs, were redesigned considering the before mentioned transistor stack limitation. Additionally, because of the long distances traveled by many signals in our chips, very strong inverters and buffers occupying up to eight rows were designed.

After the design of the schematics for all of the cells extending the trimmed library, timing characterization was performed on these cells using the tools *Cadence Encounter Library Characterizer*, and its more up to date version *Cadence Liberate*. Voltages from $0.4V$ to $1.2V$ with $100mV$ steps were used, and for each of these voltages, characterization of these cells was done on five different corners. Three different corners for the *nfet-pfet* transistors characterization were used, *FAST-FAST* (*ff*), *SLOW-SLOW* (*ss*) and *TYPICAL-TYPICAL* (*tt*). The temperatures used were $-40C$, $125C$ and $27C$. Three different percentages 90%, 100% and 110% were used for every considered voltage. A table featuring the propagation delay of a standard inverter for different voltage supplies is shown in Table 6.1. The load capacitance used was $4fF$ which corresponds roughly to the input capacitance of four of the analyzed inverters. The objective of these tables is to give an idea to the reader, the speeds one can achieve at different voltage supplies, and how corners, as voltage is lowered, display more relative variation.

Voltage	0.90V,ss 125C	0.90V,ss -40C	1.00V,tt 27C	1.10V,ff 125C	1.10V,ff -40C	Relative difference with respect to typical corner				
0.4V R	10.240ns	1075.560ns	19.726ns	1.454ns	22.046ns	51.91%	5452.47%	100.00%	7.37%	111.76%
0.4V F	2.920ns	42.489ns	2.735ns	0.949ns	2.156ns	106.74%	1553.40%	100.00%	34.68%	78.82%
0.5V R	2.685ns	77.174ns	2.680ns	0.368ns	1.575ns	100.17%	2879.43%	100.00%	13.72%	58.75%
0.5V F	0.922ns	5.059ns	0.740ns	0.227ns	0.481ns	124.53%	683.37%	100.00%	30.71%	65.04%
0.6V R	0.829ns	7.025ns	0.571ns	0.162ns	0.300ns	145.33%	1230.89%	100.00%	28.46%	52.49%
0.6V F	0.392ns	1.276ns	0.270ns	0.104ns	0.137ns	145.12%	471.83%	100.00%	38.59%	50.77%
0.7V R	0.337ns	1.073ns	0.211ns	0.097ns	0.118ns	159.75%	507.99%	100.00%	46.15%	55.73%
0.7V F	0.200ns	0.399ns	0.118ns	0.061ns	0.060ns	169.09%	336.96%	100.00%	51.75%	50.30%
0.8V R	0.181ns	0.312ns	0.118ns	0.069ns	0.068ns	152.38%	263.56%	100.00%	58.41%	57.78%
0.8V F	0.115ns	0.150ns	0.064ns	0.042ns	0.036ns	179.23%	232.20%	100.00%	65.08%	56.59%
0.9V R	0.117ns	0.144ns	0.075ns	0.054ns	0.049ns	155.01%	190.57%	100.00%	71.83%	64.78%
0.9V F	0.074ns	0.074ns	0.043ns	0.032ns	0.027ns	174.19%	174.17%	100.00%	75.33%	63.47%
1.0V R	0.085ns	0.087ns	0.057ns	0.045ns	0.039ns	150.33%	153.75%	100.00%	79.40%	68.30%
1.0V F	0.053ns	0.047ns	0.032ns	0.027ns	0.022ns	163.81%	145.87%	100.00%	82.53%	68.97%
1.1V R	0.068ns	0.063ns	0.046ns	0.039ns	0.033ns	146.01%	135.11%	100.00%	84.60%	71.04%
1.1V F	0.041ns	0.035ns	0.026ns	0.023ns	0.019ns	153.93%	131.01%	100.00%	87.46%	72.99%
1.2V R	0.056ns	0.049ns	0.040ns	0.035ns	0.029ns	142.35%	124.73%	100.00%	88.29%	73.18%
1.2V F	0.033ns	0.028ns	0.023ns	0.021ns	0.017ns	145.92%	122.33%	100.00%	90.96%	75.96%

Table 6.1: Single inverter *SEN_INV_1* propagation delay. Delay considered for nine different voltages. For each of those voltages five corners were calculated. The last five columns help to visualize how four corners deviate from the typical corner. R stands for rising and F stands for falling.

6.3 SRAM Library Design

A lot of work has been done in the design of processors in the subthreshold or near-subthreshold region,[38–41] and even approaches have been taken where the supply voltage can be adaptively changed.[42] But one very important aspect of processors is that generally they require cache memory, and the area taken by that memory is generally more than the one taken by the actual processor. In the case of this project, where different PUs will perform different types of processing, local cache to those PUs is a necessity. The memory surrounding a processor usually takes the majority of the switching power, as well as in the case of static power dissipation. It is for this reason that different architectures for SRAM cells were considered, and some common techniques were reviewed for the design of a SRAM cell library.[43] The more tempting approach was the one presented in,[44] where a specific 9T SRAM cell is presented for $65nm/55$nm process, which is exactly what is needed. Sizes for the SRAM cell transistors are carefully chosen and an operation down to 300mV has been reported.

An SRAM library was then designed due to the usage of local storage in almost every single PU type. The SRAM cell schematic used is the one shown in Figure 6.2. In Figure 6.3, on the left, the layout of the cell is presented, and on the right the same layout with only diffusion and polysilicon is shown. The shown cell actually contains two SRAM cells, and the reason for this design is that more compactness was reached. The reasons for the sizes of the transistors are mainly because of stability enhancement, leakage current reduction and noise reduction.[44]

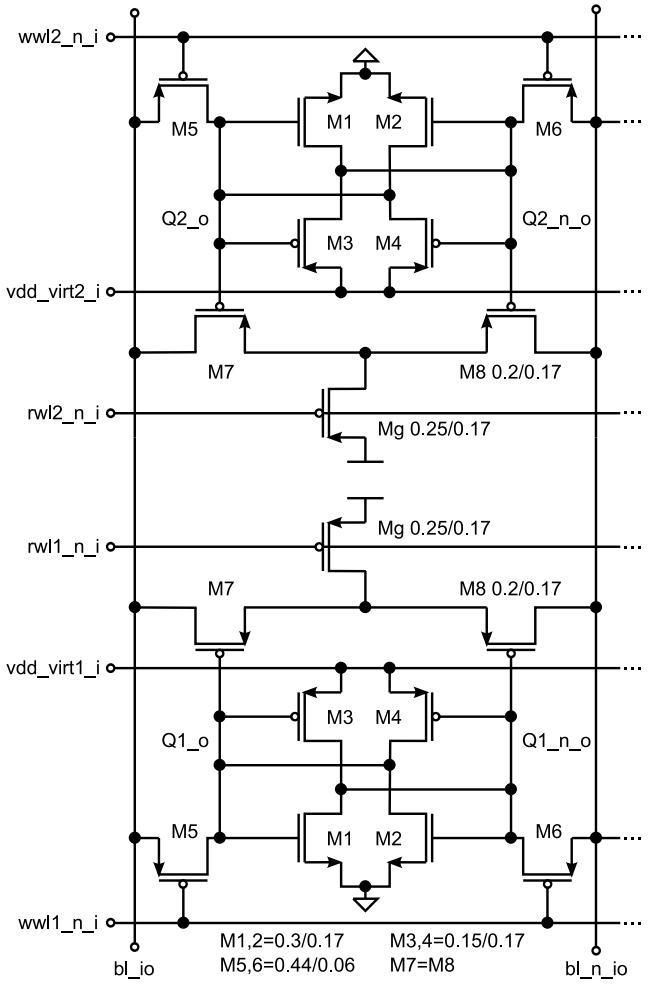


Figure 6.2: SRAM cell schematic. Due to compactness, two SRAM cells were put together in the basic SRAM cell.

One of the most recognizable characteristics of this 9T SRAM cell is that the sensing of the memory bits does not modify the state of state holding nets, when compared to SRAM cells using lower number of transistors. This makes the cell more immune to read noise when operating at very low voltages. An additional feature in this cell is the presence of the *VDD_virt_i* input. This input is the one providing power to the back-to-back inverter pair. The main objective of this input is to be

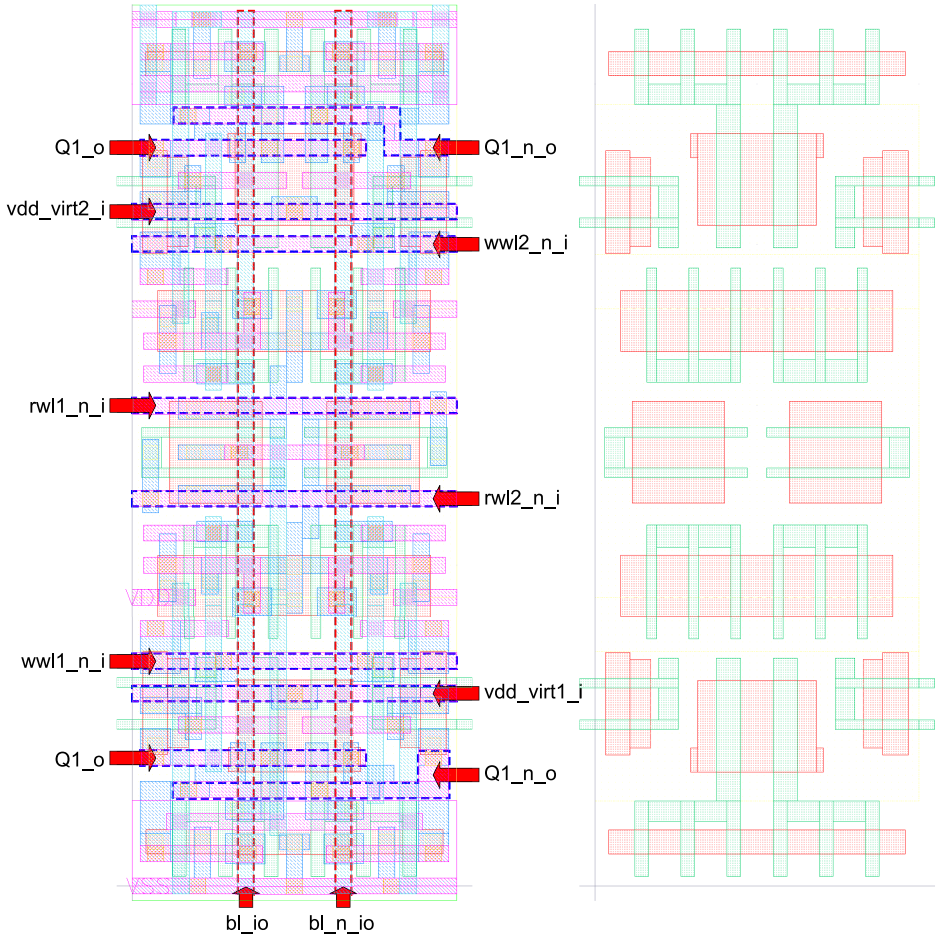


Figure 6.3: SRAM cell layout. On the left the full layout of the two SRAM cells. On the right only the polysilicon and diffusion layers are shown.

lowered to ground when writing the cell. This would reduce the fight these pair of inverters put when their held value needs to be changed, saving power.

A library of several SRAM sizes was developed using the mentioned cell. These sizes go, for the number of words, from 64 to 512 in power of 2, and each word can hold 8, 16, 24 or 32 bits, making a total of 16 different SRAM memories. Layout for these memories were designed using the same rail height as the regular standard cells. This characteristic allowed these SRAMs to be considered an extension of the

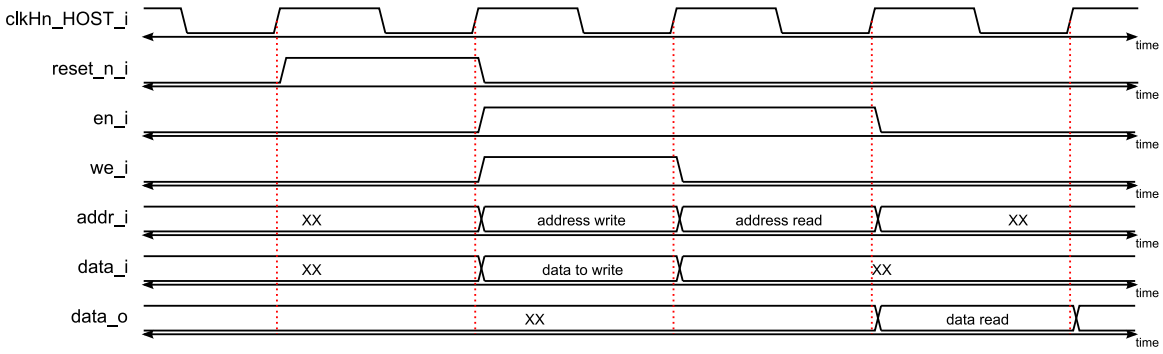


Figure 6.4: SRAM timing diagram. Reset, one write operation and one read operation are performed.

used standard cell library, so that no power ring for these memories is needed when synthesizing an architecture, allowing to gain in area. Figure 6.4 depicts the timing diagram for resetting the memory, and performing a write and a read operation. The purpose of this reset signal is to make sure that the asynchronous driver controlling internally the memory, starts from a correct state. The reset transaction needs to be done only once after power up. This reset input is latched, so it acts with a one clock cycle delay. It can be observed that data is read and written in memory in the same fashion as it is done in *Xilinx* block rams. This was designed on purpose so that tested FPGA architectures using Xilinx BRAMs could be easily ported to an *asic*.

The overall architecture for a every SRAM memory is presented in Figure 6.5. Right from the beginning two options were given, all the internal operations in the memory could be aligned with a clock signal, or a self-driven architecture using asynchronous logic could be designed. This logic would run faster, but at the expense of building more complicated circuits. Bitlines in the memory need to be pre-charged to ground before any write or read operation, and it is after this that some of those lines

are pulled up either by the selected memory word in the case of reading, or by the block *Write Logic Full* in the case of writing. The asynchronous option was finally decided upon. The *Async Control* block is the asynchronous driver that controls the memory internally. It is from this block that all the control signals are sent out to the different memory blocks. When a read or write operation is received (*en_i='1'* and *we_i='0'* for read, and *en_i='1'* and *we_i='1'* for write), the *pc* signal is set to '1' making the *Precharge Full* block discharge all the bit lines (*bl* and *bl* $\sim$ lines) to ground. After this, the *SRAM feedback* block senses when all of the bit lines are discharged, and then a '1' is sent to the asynchronous driver through the *nor* signal to let it know that the read or write operation can take place. This is done by performing a big NOR operation on all the bit lines. To make sure that the bitline voltage values are close to ground, Schimidt Trigger inverters were used. These inverters will trigger a '1' for the *nor* signal only when all the bitline voltages are close to ground.

Figure 6.6 depicts the schematic of the bias-less current-based sense amplifier used for every bit line with its negated counterpart. Inputs *bl_i* and *bl_n_i* are a bit line and its negated value. Input *enable_i* will enable the two current mirrors at the bottom of the schematic. Before performing a read operation, the *latch_i* input signal will be at '1', shorting both *Q_o* and *Q_n_o* signals together. When a read operation is performed, *latch_i* will transition to '0' allowing the two current sources to decide where the cross-coupled *pfet* transistors should tilt signals *Q_o* and *Q_n_o* to. Before reading, both *Q_o* and *Q_n_o* nets will be at '0', making *cntrl = '0'*. This will set

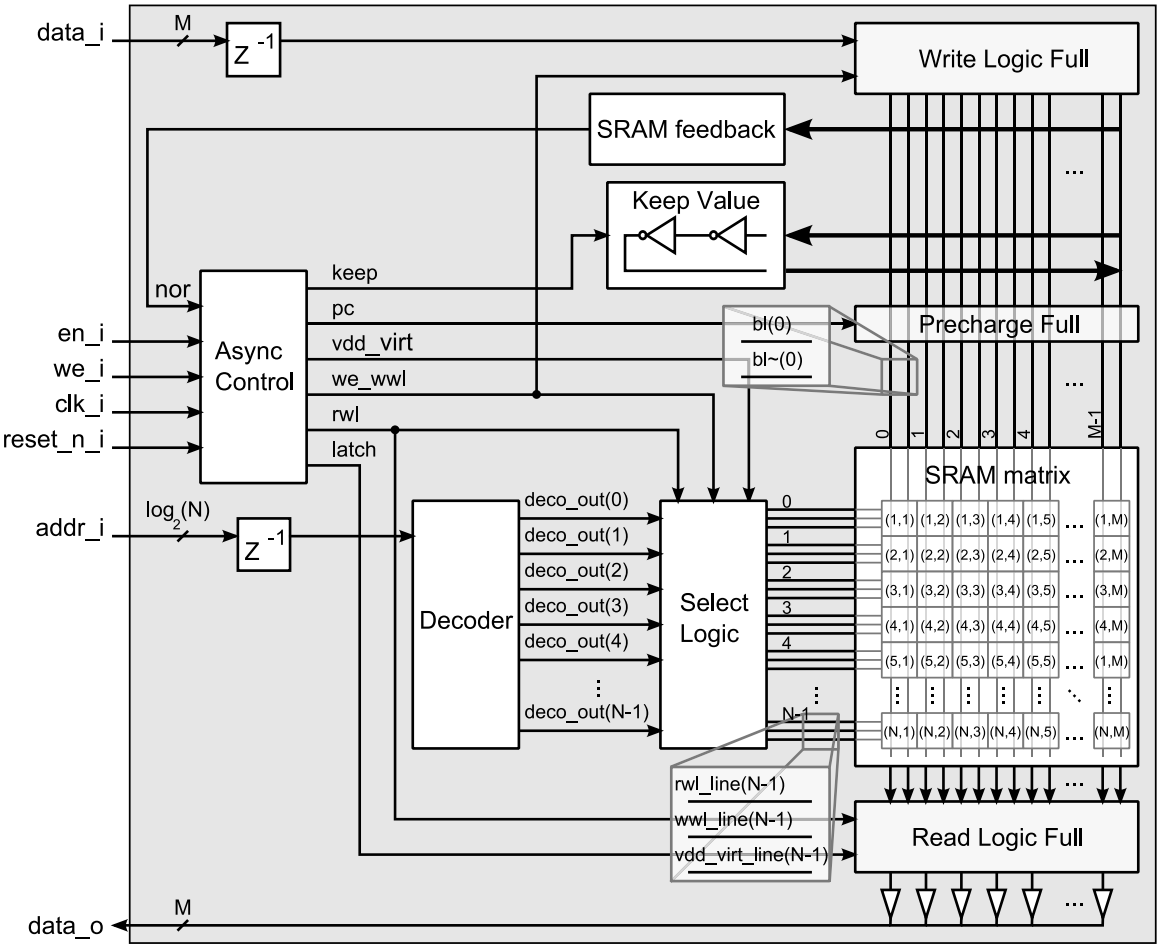


Figure 6.5: SRAM architecture diagram. Blocks making up the architecture of every SRAM memory. M is the number of bits in a word, and N is the maximum number of words.

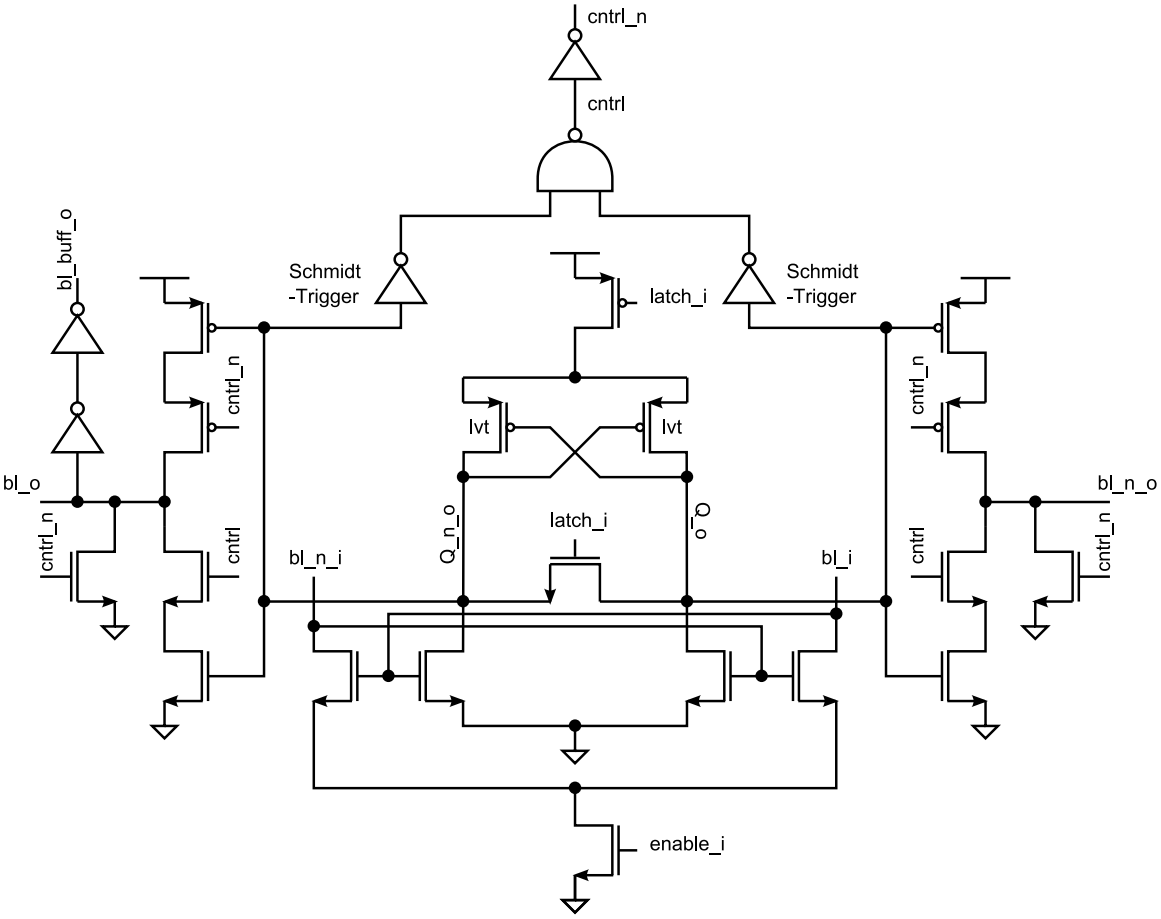


Figure 6.6: SRAM current-based sense amplifier schematic.

outputs *bl_o* and *bl_n_o* to ground. When one of the Schmidt-Trigger inverters starts sensing a change, then the *cntrl* and *cntrl_n* signals are updated, allowing outputs *bl_o* and *bl_n_o* output the read values. Output *bl_buff_o* is the output that will be directed to the SRAM output data port.

Figure 6.7 shows a diagram with the different blocks building the asynchronous driver. Figure 6.8 shows the state diagrams for both *qw* and *qr* signals. If a reset command is received then both *qw* and *qr* will go to zero. If a read command is received, then *qr* will switch, and if on the other hand a write command is received

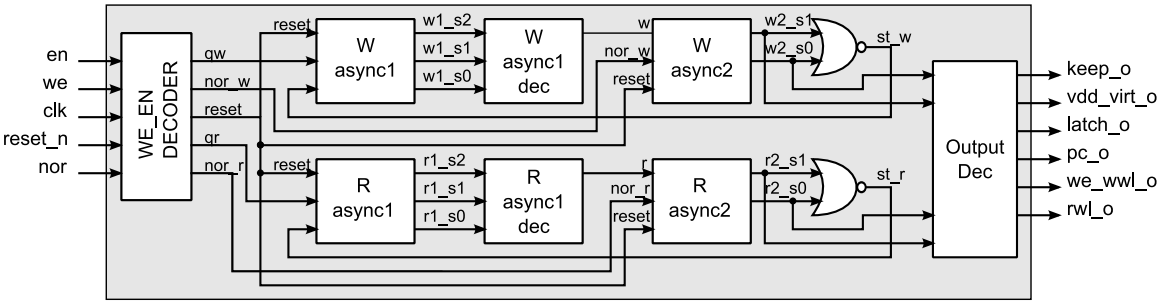


Figure 6.7: Diagram of the blocks composing the SRAM asynchronous driver. Blocks with *dec* do not posses any asynchronous logic, they are just decoders.

qw will. It is this switch that will be used in the asynchronous logic to trigger all of the internal operations of the memory. The architecture in Figure 6.7 is divided into two rows, the ones performing the operations for when a read command is received, and the ones for the write command. They are identified by the usage of R or W in their block names. Figure 6.9 show how the different *async* blocks work. For the case of the *async1* block, the inputs are c and st, and for the *async2* blocks, r and Nor are the inputs.

A more in detail functioning of the memory driver for both the cases when a word is read and a word is written is now introduced.

Read operation:

1. Signal *qr* transitions high when *en_i = '1'* and *we_i = '0'* in Figure 6.7.

2. Internal state *Rstate1* in block *R async1* transitions from *R0* to *R01*, making signal r go high.

3. The change in signal r makes internal state *Rstate2* in block *R async2* transi-

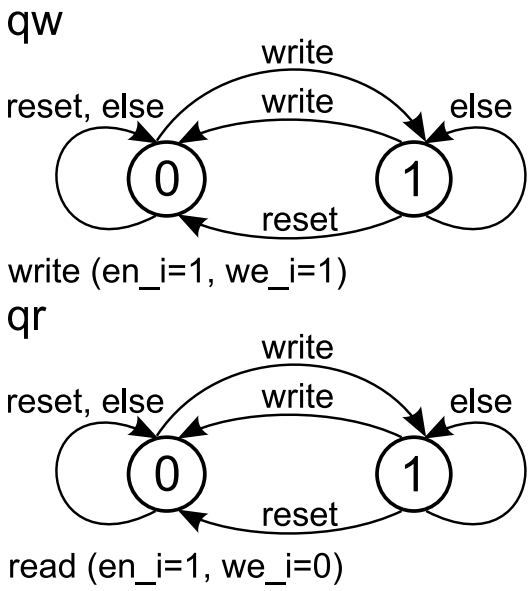


Figure 6.8: State diagram for signals *qw* and *qr* in Figure 6.7.

tion from *R0* to *R1*, making the signal *latch* and *st_r* transition high and low respectively. Signal *latch* will reset the sense amplifiers, so that when the *latch* signal goes back low, the sensing of the bitlines will be performed and a word value will be sent to the output *data_o* in Figure 6.5.

4. The change in signal *st_r* will make *Rstate1* transition from *R01* to *R02* making signal *r* transition low.

5. The last change will make *Rstate2* change from *R1* to *R2* and signal *pc* will be set high. When this signal is set high, block *Precharge Full* from Figure 6.5 will discharge all of the bitlines to ground. After a little while when all of the lines are discharged, *SRAM feedback* block from Figure 6.5 will sense this and a high transition will be received through the *nor_i* input.

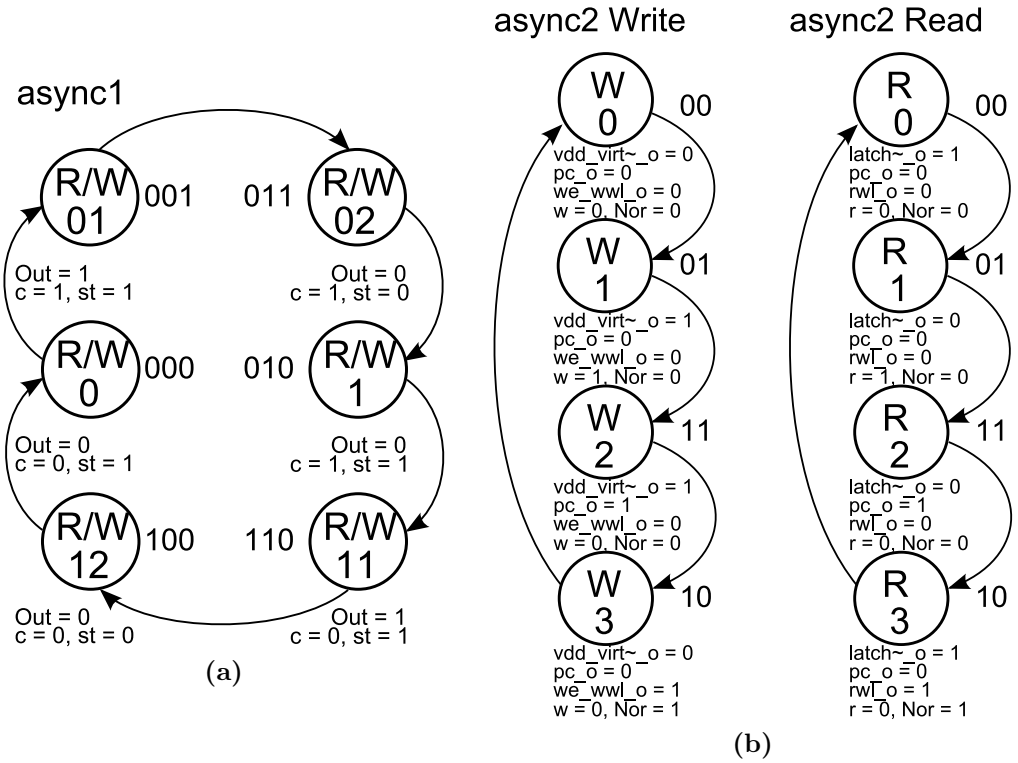


Figure 6.9: Asynchronous state diagrams for both the *async1* and *async2* blocks from Figure 6.7. Letter W stands for the state diagram for when a write is perform, and R for when a read is performed. Figure 6.9a shows an overlap of both read and write cases.

6. The change of state in *Rstate2* will make *Rstate2* internal state go from *R2* to *R3*. This change will make signal *pc* go low (this makes the block *Precharge Full* stop discharging all of the bitlines so that now some of them can be driven high in the read operation), *latch* go low (meaning that one is ready to read), and signal *rwl* will transition high (allowing the target word in memory drive the bitlines through the input *addr_i*).

7. When now the bitlines are driven by the target word in the memory array, signal *nor_i* will transition low, and then *nor_r* will consequently transition

low, making *Rstate2* change from *R3* to *R0*.

8. This last state change will make signals *st_r* and *rwl* go back to their original values, making *Rstate2* internal states transition from *R02* to *R1*. For the next time a read operation is performed, the whole process will be repeated, but now *Rstate1* will transition from *R1*, to *R11*, to *R12*, and back to the original *R0*.

An additional signal is present in Figure 6.5, the *keep* signal. This signal is set high when either of the internal states in blocks *R async2* and *W async2* are different than *R0* and *W0* respectively. The problem found was that, if a write or read command is not received with a certain frequency, bitlines will be discharged down to ground and the signal *nor_i* will be set high triggering an undesired asynchronous process. This is the reason the block *Keep Value* will sense the bitlines and try to maintain the logic value by injecting or withdrawing a small current.

Write operation:

For the write operation a similar approach is taken. In this case there are some differences in the signals that are being driven. After receiving a write operation that switches the *qw* signal, the whole process is very similar to the read operation, but instead of the *latch* and *rwl* signals, signals *vdd_virt* and *we_wwl* are driven. The signal *vdd_virt* is a signal that powers off the target word in memory so that the write operation performed when signal *we_wwl* transitions high can be done utilizing less power.

Figure 6.10 presents a detailed timing diagram of all the signals involved in the asynchronous control unit for the SRAM memories. The layout designed for the $64x32$ SRAM memory is shown in Figure 6.11. One additional comment is worth to mention, and that is that all of the SRAM memory cells were all designed using only up to metal three, meaning that the only metals used were *M1*, *M2* and *M3*. This is a very convenient characteristic, as it allows to use all of the remaining metals to perform routing on top of the memories. The memory shown in Figure 6.11 achieves a $10.55\mu m^2$ of area per bit, with an operation down to 400mV.

6.4 SRAM Test Chip GF5

A chip named GF5, with a size of $3.5mmx3.5mm$, was fabricated with the same $55nm$ Global Foundries process for the test of several architectures. One of them was the SRAM cell library. In previous tapeouts, the custom designed pads where not incorporated as part of the standard cell library, meaning that signals had to manually routed to the pads from the chip cores. In this new chip, the pads were incorporated in the flow, and then manual routing of the connections to the pads was not required, reducing significantly the likelihood of human errors.

Two types of characterization are needed when blocks, pads for instance, are desired to be incorporated in the synthesis flow, the abstract characteristics of the cell (a file that contains all the geometrical information of the block so that the synthesis

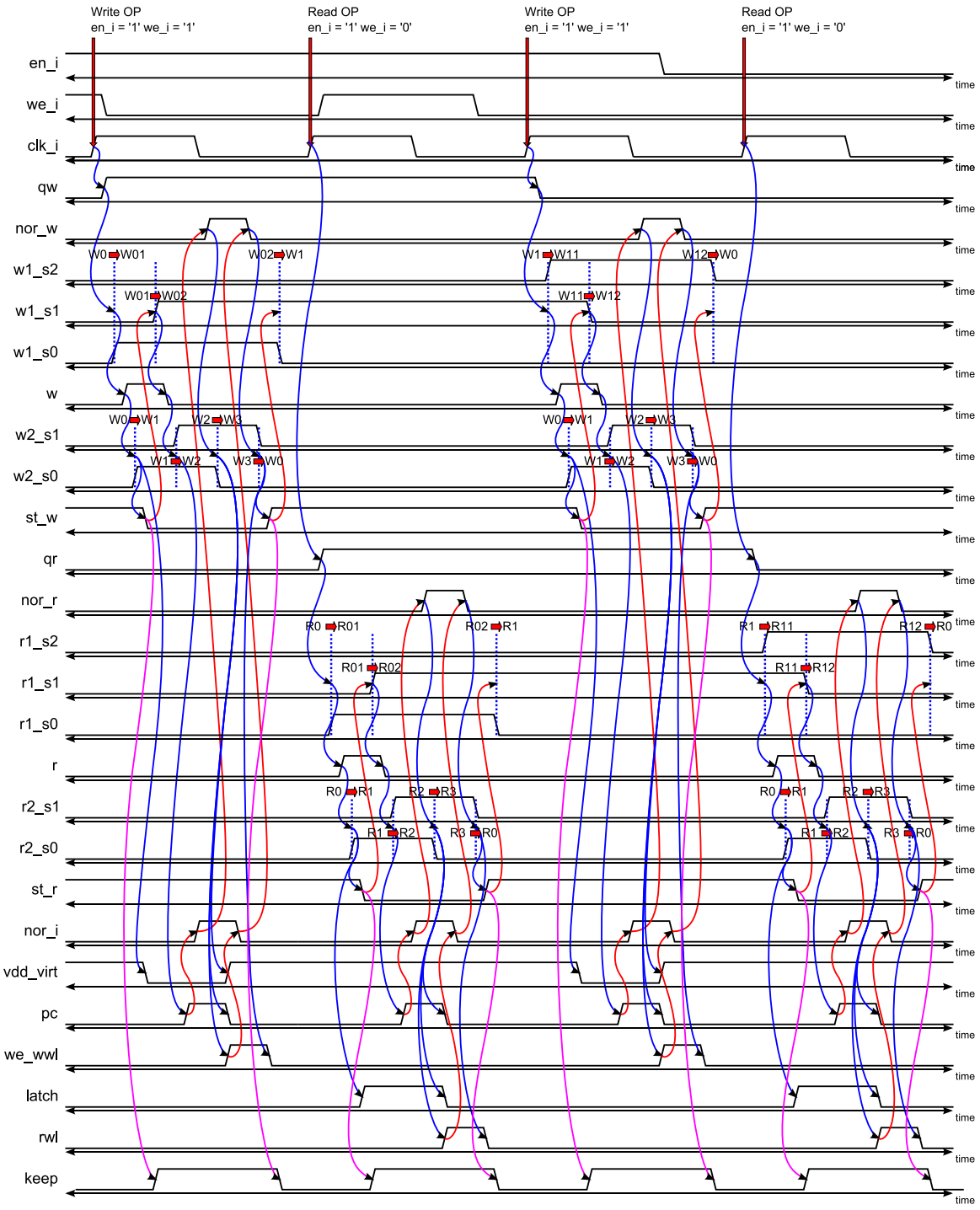


Figure 6.10: Asynchronous driver timing diagram. Detailed timing diagram of all the signals in the SRAM asynchronous controller.

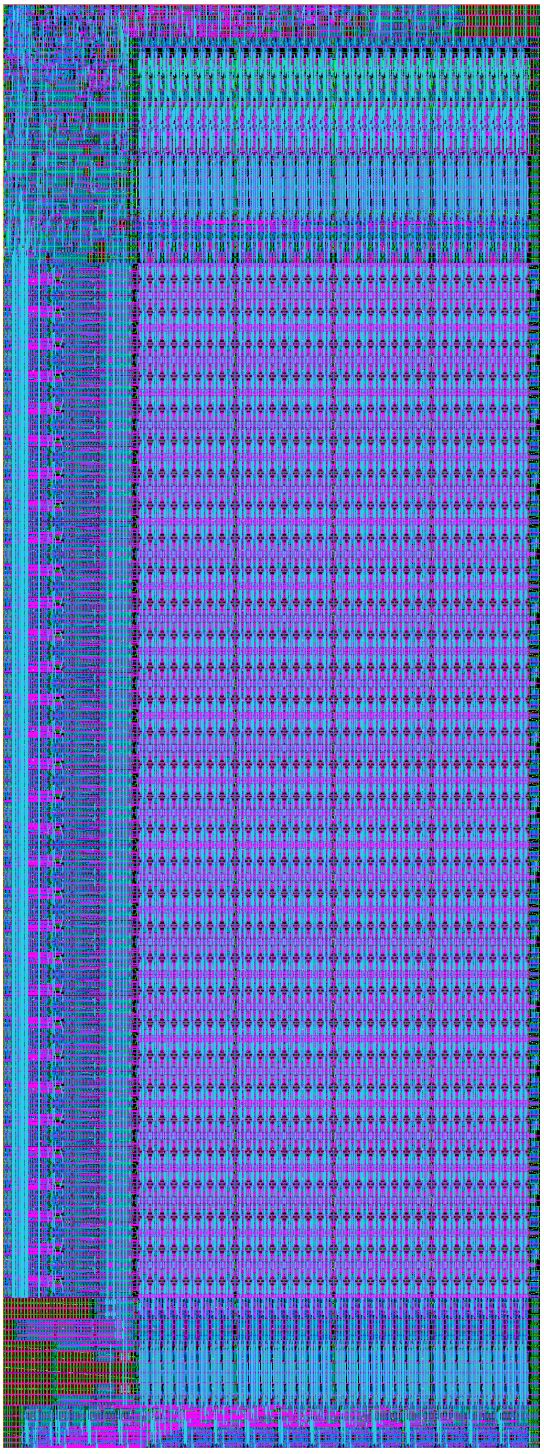


Figure 6.11: Layout for the 64x32 SRAM block. The 64x32 SRAM memory cell is one of the 16 different SRAM memory cells. Only up to metal three is used for all of the SRAM memories.

tool knows its dimensions, where to route, etc.) and the timing characteristics (required for achieving speed constraints in our design). In this chip the first type of characterization was done accurately, but the second one, due to lack of time, was reduced to a simplified version where the input and output pads are considered to have the same timing characteristics as one of the buffers in our standard cell library.

In this GF5 test chip, a single pad frame was used. The idea was to have a top-level synthesis of the chip where the different cores are placed as blocks. So that a better use of area could be achieved, it was desired to share the logical pads among all the cores. For the case of the input pads to the chip, these pads can be routed to all of the cores without any complication. But for the case of the output pads, in order not to use additional pads as multiplexer control signals, a logical OR for all of the outputs was decided to be performed. This strategy for the output pads would only work if only one core is powered at a time, and that is the reason that dedicated power pads for each of the cores were incorporated. For the case of biases, when possible, pads were also shared among cores. These ideas can be seen in Fig 6.12.

There are seven different cores in the chip, the CID core (designed by Gaspar Tognetti), the ACM core (designed by Philippe Pouliquen), the VVM core (designed by Kayode Sanni), the MORPH core (designed by Martin Villemur), the PLL core (IBM design), the IFAT core (designed by Jamal Molin) and an SRAM core (designed by Tomás Figliolia). Each of the cores works at a unique voltage with exception of

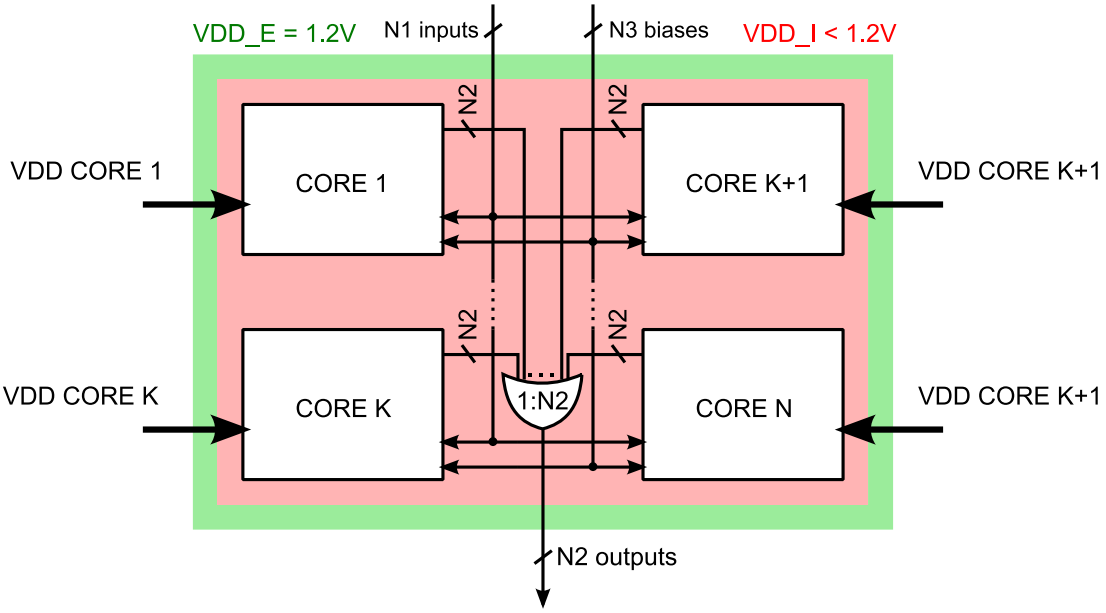


Figure 6.12: Pad sharing in GF5 chip. In green, the voltage used by the pads in the communication to the outside world. In red, the voltage used in the top logic synthesis of the chip. Each of the big black arrows represents each of the independent core voltages. All of the $N1$ inputs are being distributed to all of the cores. The $N2$ output signals are ORed bit to bit for all of the cores. For the case of the biases, if possible, they were also shared.

the PLL core which has two voltage sources, an analog ground and a digital ground. Ground is a unique net shared among all of the cores, with the exception of the analog ground for the PLL. This ground is provided with the pads beginning with VSS. Power pads for VDD_I and VDD_E start with those same names. In Figure 6.13 the layout of the chip is presented. All the pad names are shown.

In Tables 6.2, 6.3 and 6.4 the sharing of all the input, output and bias pads is shown. Each of the pads in the first column is shared among all of the signals in the same row.

Even if all of the cores would not run at the same maximum speed, the top level

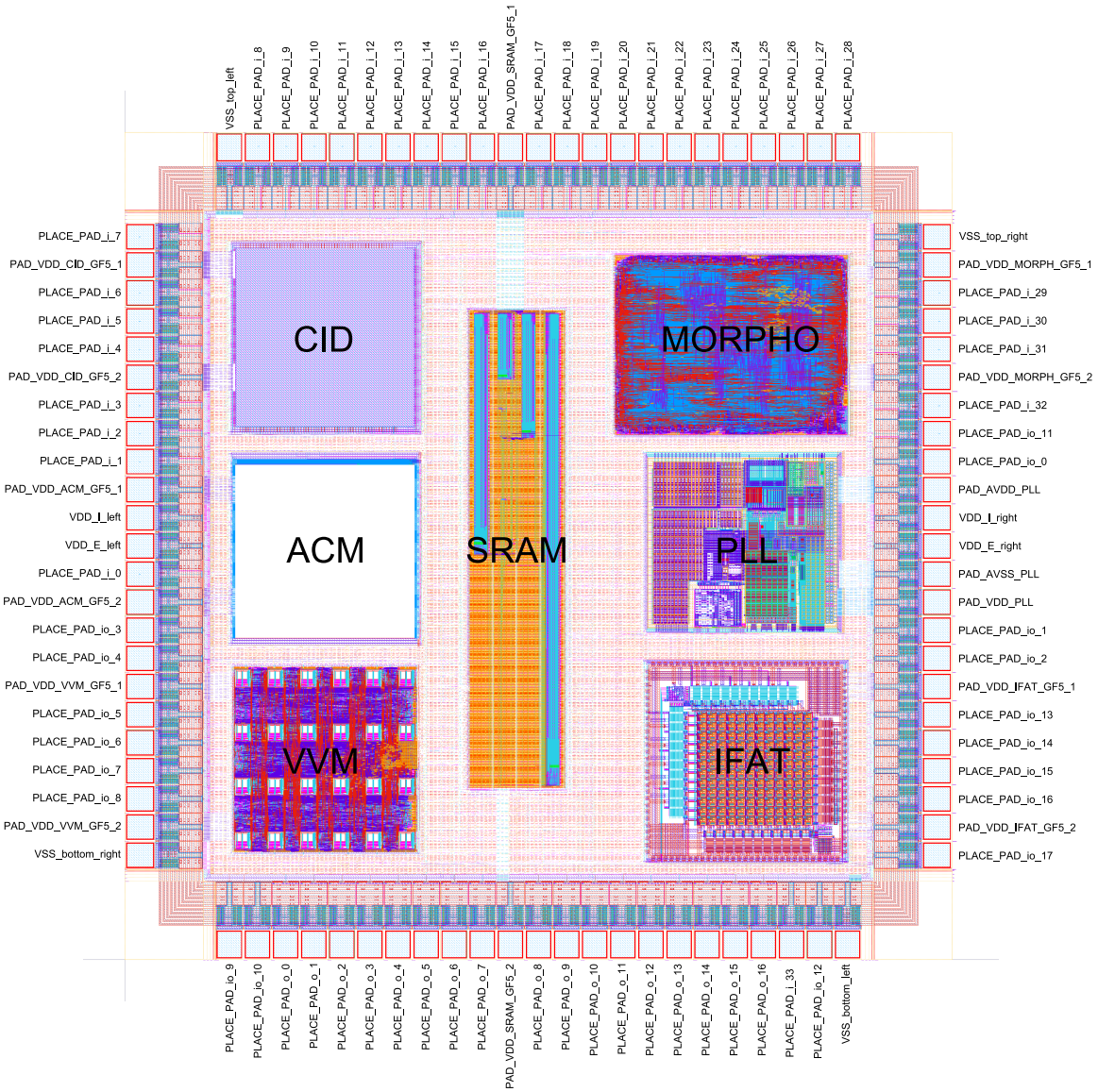


Figure 6.13: Layout view of GF5 chip.

INPUT	IFAT	PLL	ACM	CID	MORPHO	VVM	SRAM	OSC
PLACE_PAD_i_0	Gclk_i	REFCLK_i	clock1_i	port_i[0]	clk_ph0	clk_i	clk_i	-
PLACE_PAD_i_1	W_i[0]	FBKCLK_i	clock2_i	port_i[1]	rst	rst_i	en_i	-
PLACE_PAD_i_2	W_i[1]	CLK_i	data_i[0]	port_i[2]	sel	en_i	we_i	-
PLACE_PAD_i_3	Rst_Array_i	INTFBK_i	data_i[1]	port_i[3]	address[0]	bus_i[0]	reset_n_i	-
PLACE_PAD_i_4	Ren_i	SR_i	data_i[2]	port_i[4]	address[1]	bus_i[1]	addr_i[0]	-
PLACE_PAD_i_5	CellReset_i	RESET_i	data_i[3]	port_i[5]	address[2]	bus_i[2]	addr_i[1]	-
PLACE_PAD_i_6	Rst_Rcvr_i	BYPASS_i	data_i[4]	port_i[6]	address[3]	bus_i[3]	addr_i[2]	-
PLACE_PAD_i_7	data_i[0]	STOPCLKA_i	data_i[5]	port_i[7]	address[4]	bus_i[4]	addr_i[3]	-
PLACE_PAD_i_8	data_i[1]	STOPCLKB_i	data_i[6]	port_i[8]	address[5]	bus_i[5]	addr_i[4]	-
PLACE_PAD_i_9	data_i[2]	SLEEP_i	data_i[7]	port_i[9]	address[6]	bus_i[6]	addr_i[5]	-
PLACE_PAD_i_10	data_i[3]	DLT_i	data_i[8]	port_i[10]	address[7]	bus_i[7]	addr_i[6]	-
PLACE_PAD_i_11	data_i[4]	-	data_i[9]	port_i[11]	address[8]	bus_i[8]	addr_i[7]	-
PLACE_PAD_i_12	data_i[5]	-	data_i[10]	port_i[12]	address[9]	bus_i[9]	addr_i[8]	-
PLACE_PAD_i_13	data_i[6]	-	data_i[11]	port_i[13]	wr_en	bus_i[10]	addr_i[9]	-
PLACE_PAD_i_14	data_i[7]	-	data_i[12]	port_i[14]	data_in[0]	bus_i[11]	data_i[0]	-
PLACE_PAD_i_15	data_i[8]	-	data_i[13]	port_i[15]	data_in[1]	bus_i[12]	data_i[1]	-
PLACE_PAD_i_16	data_i[9]	-	data_i[14]	port_i[16]	data_in[2]	bus_i[13]	data_i[2]	-
PLACE_PAD_i_17	data_i[10]	-	data_i[15]	port_i[17]	data_in[3]	bus_i[14]	data_i[3]	-
PLACE_PAD_i_18	data_i[11]	-	addr_i[0]	port_i[18]	data_in[4]	bus_i[15]	data_i[4]	-
PLACE_PAD_i_19	Rst_Xmit_i	-	addr_i[1]	port_i[19]	data_in[5]	bus_sel_i	data_i[5]	-
PLACE_PAD_i_20	XmitAck_i	-	addr_i[2]	port_i[20]	data_in[6]	addr_i[0]	data_i[6]	-
PLACE_PAD_i_21	-	-	addr_i[3]	port_i[21]	data_in[7]	addr_i[1]	data_i[7]	-
PLACE_PAD_i_22	-	-	addr_i[4]	port_i[22]	-	addr_i[2]	data_i[8]	-
PLACE_PAD_i_23	-	-	addr_i[5]	port_i[23]	-	addr_i[3]	data_i[9]	-
PLACE_PAD_i_24	-	-	addr_i[6]	port_i[24]	-	addr_i[4]	data_i[10]	-
PLACE_PAD_i_25	-	-	addr_i[7]	port_i[25]	-	addr_i[5]	data_i[11]	-
PLACE_PAD_i_26	-	-	addr_i[8]	port_i[26]	-	addr_sel_i	data_i[12]	-
PLACE_PAD_i_27	-	-	write_enable_i	port_i[27]	-	opcode_i[0]	data_i[13]	-
PLACE_PAD_i_28	-	-	sample_i	port_i[28]	-	opcode_i[1]	data_i[14]	-
PLACE_PAD_i_29	-	-	select_i[0]	port_i[29]	-	opcode_i[2]	data_i[15]	-
PLACE_PAD_i_30	-	-	select_i[1]	port_i[30]	-	-	-	-
PLACE_PAD_i_31	-	-	select_i[2]	port_i[31]	-	-	-	-
PLACE_PAD_i_32	-	-	select_i[3]	port_i[32]	-	-	-	-
PLACE_PAD_i_33	-	-	-	-	-	-	-	OSC_i

Table 6.2: GF5 chip input pads. Description of how input pads are shared among all of the tested blocks.

INPUT	IFAT	PLL	ACM	CID	MORPHO	VVM	SRAM	OSC
PLACE_PAD_o_0	data_o[0]	PLLOUTA_o	data_o[0]	port_o[0]	ready_out	bus_o[0]	data_o[0]	-
PLACE_PAD_o_1	data_o[1]	PLLOUTB_o	data_o[1]	port_o[1]	data_out[0]	bus_o[1]	data_o[1]	-
PLACE_PAD_o_2	data_o[2]	PLLSYNCA_o	data_o[2]	port_o[2]	data_out[1]	bus_o[2]	data_o[2]	-
PLACE_PAD_o_3	data_o[3]	PLLSYNCB_o	data_o[3]	port_o[3]	data_out[2]	bus_o[3]	data_o[3]	-
PLACE_PAD_o_4	data_o[4]	OBSERVE0_o	data_o[4]	port_o[4]	data_out[3]	bus_o[4]	data_o[4]	-
PLACE_PAD_o_5	data_o[5]	OBSERVE1_o	data_o[5]	port_o[5]	data_out[4]	bus_o[5]	data_o[5]	-
PLACE_PAD_o_6	data_o[6]	TESTOUTFREQ_o	data_o[6]	port_o[6]	data_out[5]	bus_o[6]	data_o[6]	-
PLACE_PAD_o_7	data_o[7]	TESTOUTLOCK_o	data_o[7]	port_o[7]	data_out[6]	bus_o[7]	data_o[7]	-
PLACE_PAD_o_8	data_o[8]	CE0ASST_o	data_o[8]	port_o[8]	data_out[7]	bus_o[8]	data_o[8]	-
PLACE_PAD_o_9	data_o[9]	DIVA_O	data_o[9]	port_o[9]	-	bus_o[9]	data_o[9]	-
PLACE_PAD_o_10	data_o[10]	DIVB_O	data_o[10]	port_o[10]	-	bus_o[10]	data_o[10]	-
PLACE_PAD_o_11	data_o[11]	-	data_o[11]	port_o[11]	-	bus_o[11]	data_o[11]	-
PLACE_PAD_o_12	RcvrAck_o	-	data_o[12]	port_o[12]	-	bus_o[12]	data_o[12]	-
PLACE_PAD_o_13	-	-	data_o[13]	port_o[13]	-	bus_o[13]	data_o[13]	-
PLACE_PAD_o_14	-	-	data_o[14]	port_o[14]	-	bus_o[14]	data_o[14]	-
PLACE_PAD_o_15	-	-	data_o[15]	port_o[15]	-	bus_o[15]	data_o[15]	-
PLACE_PAD_o_16	-	-	-	-	-	-	-	OSC_o

Table 6.3: GF5 chip output pads. Description of how output pads are shared among all of the tested blocks.

INPUT	IFAT	PLL	ACM	CID	MORPHO	VVM	SRAM	OSC
PLACE_PAD_io_0	E_io	-	-	-	-	-	-	-
PLACE_PAD_io_1	Vrst_io	-	-	-	-	-	-	-
PLACE_PAD_io_2	Vthresh_io	-	-	-	-	-	-	-
PLACE_PAD_io_3	-	-	vbf_io	-	-	-	-	-
PLACE_PAD_io_4	-	-	vcl_io	-	-	-	-	-
PLACE_PAD_io_5	-	-	vfb_io	bias_i[2]	-	-	-	-
PLACE_PAD_io_6	-	-	-	-	-	V_inp_io	-	-
PLACE_PAD_io_7	-	-	-	-	-	V_inm_io	-	-
PLACE_PAD_io_8	-	-	-	-	-	V_fbp_io	-	-
PLACE_PAD_io_9	-	-	-	-	-	V_fbm_io	-	-
PLACE_PAD_io_10	-	-	-	-	-	V_cmi_io	-	-
PLACE_PAD_io_11	-	-	-	-	-	V_cmo_io	-	-
PLACE_PAD_io_12	-	-	-	-	-	V_b_io	-	-
PLACE_PAD_io_13	-	-	-	bias_i[3]	-	-	-	-
PLACE_PAD_io_14	-	-	-	bias_i[4]	-	-	-	-
PLACE_PAD_io_15	Vbn_io	-	vb_io	bias_i[1]	-	I_amp_io	-	-
PLACE_PAD_io_16	-	-	voa_io	bias_i[0]	-	I_cmp_io	-	-
PLACE_PAD_io_17	-	-	-	bias_i[5]	-	-	-	-

Table 6.4: GF5 chip bias pads.

synthesis was aimed to run at 1GHz considering the first four input pads in Table 6.2 as clocks, so that all of the cores could reach their maximum frequency of operation if desired. As well as the pads, all of the cores are considered blocks for which the two characterizations mentioned before had to be performed as well. Only the SRAM, VVM and MORPH cores were accurately characterized in timing. The reason for this is that each of these blocks were synthesized and then the *Cadence Innovus* tool can provide the timing model automatically. For the other blocks, the designs were all custom, and due to lack of time the timing characterization for them was not performed.

Apart from all of the mentioned cores, one input and one output pad were utilized for testing the maximum frequency of operation for the custom-designed pads. The signal received from an input pad is inverted and fed to the output pad. By doing this, when shorting at the bondpad level the input and output pads, an oscillation is achieved, and the frequency of oscillation will determine the maximum frequency of operation for the pads. These pads are $PLACE\_PAD\_i\_33$ and $PLACE\_PAD\_o\_16$.

The SRAM memory blocks tested in this chip were only the ones with a word size of 16bits. The impossibility of placing all the other cells was due to the lack of space and the lack of input pads. As it can be seen from Table 6.2 and 6.3, the input and output word is 16 bits long. Input *reset_n_i* represents the inverted reset input, *en_i* is the input enabling a read or write operation, *we_i* is the input determining the type of operation, and *clk_i* is the clock input. The maximum size SRAM memory block

stores 512 words, and then nine bits would be required to address each of the words. A 10-bit address input *addr_i* was used in the design. The additional bit allowed to address one of the four available SRAM memories. If *addr_i(9) = '1'*, then the 512 words memory is addressed, if *addr_i(9 downto 8) = "01"* the 256 words memory is addressed, if *addr_i(9 downto 7) = "001"* the 128 words memory is addressed, and if finally *addr_i(9 downto 6) = "0001"* then the 64 words memory is the one being addressed.

All of the SRAM memories in GF5 were tested by Jonah Segupta, and the maximum successful clock speeds found are presented in Table 6.5. Because none of the ORed outputs from the different tested blocks were pipelined in any way in their way to the output pads, speeds for the SRAM memories are supposed to be faster than the ones presented in Table 6.5. Tests on the SRAM memory blocks were done lowering the power supply down to 600mV with a successful operation.

Memory Size	Maximum Measured Clock Speed
64	374MHz
128	295MHz
256	211MHz
512	136MHz

Table 6.5: SRAM memory maximum clock frequency. Maximum clock frequency measured for the four tested SRAM memory blocks in GF5 chip.

Chapter 7

A Stochastic Architecture for the Adams/McKay Online Change Point Detection

7.1 Introduction

In the search of algorithms to perform background-foreground segmentation on images as part of the CMPs' image processing flow, different approaches were analyzed, like the ones presented in.[45–47] One particular algorithm[48] was found to be extremely interesting as it recurrent nature, and simplicity of operations, allowed to be ported into a novel way of performing computations known as stochastic computing. This approach will be seen later in this section with the fabrication of three test

chips.

Change Point Analysis (CPA) also known as Change Point Detection (CPD) is the identification of sudden and often small changes to the parameters at the output of a system that is in the form of sequential data. Often CPA is employed for the segmentation of a signal to facilitate the process of tracking, identification or recognition. The *Bayesian*[49,50] version of a Change Point Analysis was originally described in[51] with online versions of the algorithm only recently formulated.[48,52] BOCPD a Bayesian Online Change Point Detection algorithm of Adams and McKay[48] and further advanced in[53–55] allow for online inference with causal predictive filtering processing necessary in real-time systems that interact with physical environments that can change. One of the key challenges and critique of Bayesian approaches is the high computational requirements that often necessitate high precision floating point computations.

In the process to explain the CPD algorithm, let's consider the case where a stream of independent samples x_1, x_2, ..., x_t is received. The parameters of the distribution from which these samples are drawn can suddenly change over time. If one can identify where that change of parameters occurred, then a point of change can be established. To give an example of such signal, one can consider a simple stream of zeros and ones from a serial line, where the received voltage values at the endpoint can fluctuate due to channel noise. Considering that the noise is Gaussian with mean equal to zero, the sampled signal will be distributed $\sim N(\mu, \sigma^2)$, where σ^2 is the fixed

variance of the channel noise, and μ is the actual transmitted value, for example $0V$ or $5V$. In this case, the only parameter changing over time is μ, but a model in which σ^2 changes as well, can be also considered. Samples can be distributed normally with parameters μ and σ^2, and additionally one can think that those parameters can also be drawn from another prior distribution $P(\mu, \sigma^2)$. For the transmission line example mentioned before, σ^2 can be considered fixed, but μ can be drawn from a Bernoulli distribution, with a p probability of sending $5V$, and a $(1-p)$ probability of sending $0V$.

The run-length concept is now introduced, which is the number of consecutive samples that are contemplated to have been drawn from the same distribution (the parameters' values didn't change for all of the samples in that run). At time t, the run-length will be r_t. If $r_t = k$, then the samples that are considered to be part of that run are x_{t-k}, $x_{t-(k+1)}$, ..., x_t. The number of samples from a run $r_t = k$ will be addressed as $x_t^{(r=k)}$. In Figure 7.1 a graph capturing the ideas mentioned is shown. The nodes P are the nodes that contain the parameters of the distribution from which samples x_t are taken. The change of the parameters' values P_{t-k} will trigger a reset in the count r_{t-k}, setting it to 0. On the other hand, if the parameters $P_{t-k} = P_{t-(k+1)}$ then $r_{t-k} = r_{t-(k+1)} + 1$.

The objective of this algorithm is to be able to predict, based on history, what is the probability density function $P(r_t|X_{1:t} = x_{1:t})$. Note that for this algorithm, at every time step, there is a distribution $P(r_t|X_{1:t} = x_{1:t})$, where r_t can take values from

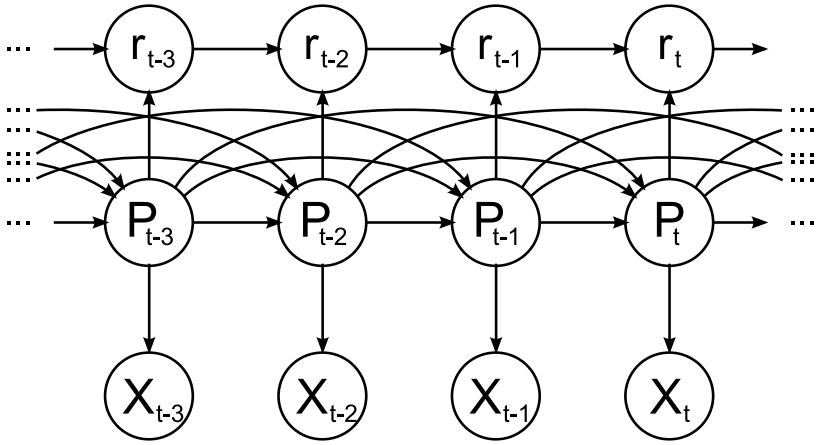


Figure 7.1: CPD algorithm graph. In this graph r represents the run-length, which can increase one count if the parameters P do not change, or it can be reset to 0 is they do. Using parameters P, a sample x is withdrawn at every time step. The current parameter value P_t can depend on all of its previous values.

0 to $t-1$. The variables r will be considered the hidden variables of this Markov chain process. There is no access to those values, and it is for this reason that a probability distribution is estimated for them. This is the probability distribution used in taking the decision that a new run has started due to the change of the parameters' values. At the bottom of Figures 2, 3 and 4 from,[48] one can see in gray-scale at each point in time the distribution $P(r_t|X_{1:t} = x_{1:t})$.

7.2 Algorithm Equation Development

A brief development of the general equation behind the CPD algorithm in[48] is presented here. The equations that will be shown are the key to understanding how the algorithm works. Since $P(r_t|X_{1:t}) \propto P(r_t, X_{1:t})$, one can first get an expression for $P(r_t, X_{1:t})$ and then normalize. The following equation results can be easily obtained

following Bayes rule.

$$\begin{aligned} P(r_t, X_{1:t}) &= \sum_{r_{t-1}} P(r_t, r_{t-1}, X_{1:t}) \\ &= \sum_{r_{t-1}} P(r_t, X_t | r_{r-1}, X_{1:t-1}) P(r_{t-1}, X_{1:t-1}) \\ &= \sum_{r_{t-1}} P(r_t | r_{t-1}, X_{1:t}) P(X_t | r_{t-1}, X_{1:t-1}) P(r_{t-1}, X_{1:t-1}) \\ &= \sum_{r_{t-1}} P(r_t | r_{t-1}) P(X_t | r_{t-1}, X_{t-1}^{(r)}) P(r_{t-1}, X_{1:t-1}) \end{aligned} \tag{7.1}$$

Notice that $P(r_t, X_{1:t})$ is a function of $P(r_{t-1}, X_{1:t-1})$. Every time a new sample arrives, by using $P(r_{t-1}, X_{1:t-1})$ from the previous time step, $P(r_t|r_{t-1})$ and $P(X_t|r_{t-1}, X_{t-1}^{(r)})$, one can obtain $P(r_t, X_{1:t})$. From,[48] distribution $P(r_t|r_{t-1})$ will be assumed to have the following form:

$$P(r_t|r_{t-1}) = \begin{cases} H(r_{t-1}+1) & \text{if } r_t = 0 \\ 1 - H(r_{t-1}+1) & \text{if } r_t = r_{t-1}+1 \\ 0 & \text{if } otherwise \end{cases} \tag{7.2}$$

Where $H(\tau)$ is the hazard function.

$$H(\tau) = \frac{P_{gap}(g = \tau)}{\sum_{t=\tau}^{\infty} P_{gap}(g = t)} \tag{7.3}$$

There is a special case in which $P_{gap}(g)$ is a discrete exponential (geometric)

distribution with timescale λ, the process is memory-less and the hazard function is constant at $H(\tau) = 1/\lambda$. This is the case considered for the developments that follow. The only thing left to do is to specify the distribution $P(X_t|r_{t-1}, X_{t-1}^{(r)})$, which is a problem that is addressed in the next subsection.

In,[48] the framework presented in Section 7.2 is shown, but no development is depicted for the different forms distribution $P(X_t|r_{t-1}, X_{t-1}^{(r)})$ can take. Development of cases in which Inverse Gamma and Normal are the forms the prior distributions over the parameters will now be presented.

7.2.1 Case of the Inverse Gamma Prior

Let's consider the following distribution:

$$\begin{aligned} P(X_t, \mu, \sigma^2|X_{t-k}, X_{t-(k-1)}...X_{t-1}) &= P(X_t, \mu, \sigma^2|X_{t-k:t-1}) \\ &= P(X_t|\mu, \sigma^2, X_{t-k:t-1})P(\mu, \sigma^2|X_{t-k:t-1}) \end{aligned}$$

Given that $X_t \sim N(\mu, \sigma^2)$ and considering all the Xs independent and identically distributes (iid), X_t depends uniquely on μ and σ^2, then:

$$\begin{aligned} P(X_t, \mu, \sigma^2|X_{t-k}, X_{t-(k-1)}...X_{t-1}) &= P(X_t|\mu, \sigma^2)P(\mu, \sigma^2|X_{t-k:t-1}) \\ &= P(X_t|\mu, \sigma^2)P(\mu|\sigma^2, X_{t-k:t-1})P(\sigma^2|X_{t-k:t-1}) \end{aligned}$$

For this distribution it is assumed that there is a random process that samples

the values of the variance σ^2 and the mean μ of the normal distribution $P(X_t|\mu, \sigma^2)$. Distribution $P(\mu|\sigma^2, X_{t-k:t-1})$ will now be considered to be a normal one with mean μ_o and variance $\sigma^2\nu$. The value of μ_o and ν will be a function of $X_{t-k:t-1}$. Additionally, $P(\sigma^2|X_{t-k:t-1})$ will be considered an inverse gamma with parameters a and b, which are also a function of $X_{t-k:t-1}$. Then:

$$P(X_t, \mu, \sigma^2|\mu_o(X_{t-k:t-1}), \nu(X_{t-k:t-1}), a(X_{t-k:t-1}), b(X_{t-k:t-1})) =$$

$$P(X_t|\mu, \sigma^2)P(\mu|\sigma^2, \mu_o(X_{t-k:t-1}), \nu(X_{t-k:t-1}))P(\sigma^2|a(X_{t-k:t-1}), b(X_{t-k:t-1})) =$$

$$P(X_t|\mu, \sigma^2)P(\mu|\sigma^2, \mu_o, \nu)P(\sigma^2|a, b) \tag{7.4}$$

Equation 7.4 hides the dependence of X_t, μ and σ^2 on $X_{t-k:t-1}$ through the parameters μ_o, ν, a and b. It is known from Bayes that:

$$P(\vec{\theta}|X) = \frac{P(X|\vec{\theta})P(\vec{\theta})}{P(X)} \tag{7.5}$$

If $\vec{\theta} = (\mu, \sigma^2)$, and $P(\vec{\theta})$ is the product of a normal distribution with a gamma inverse distribution, which is formally known as the normal inverse gamma (NIG), then:

$$P(\vec{\theta}) = P(\mu, \sigma^2) = P(\mu|\sigma^2)P(\sigma^2) = N(\mu|\mu_o, \sigma^2)\Gamma^{-1}(\sigma^2|a, b) \tag{7.6}$$

$$P(X|\vec{\theta}) = P(X_t|\mu, \sigma^2) = N(X_t, \mu, \sigma^2) \tag{7.7}$$

Since in this case $P(X|\vec{\theta})$ is normal distributed and $P(\vec{\theta})$ is NIG (Normal Inverse Gamma) distributed, then $P(\vec{\theta}|X)$ is also a NIG, since the NIG distribution is the conjugate prior for the normal distribution.

For a particular value of X_t $P(X_t = x_t, \mu, \sigma^2|X_{t-k:t-1}) \propto P(\mu, \sigma^2|X_{t-k:t-1}, X_t = x_t)$, then it can be said that $P(X_t = x_t, \mu, \sigma^2|X_{t-k:t-1}) \propto NIG(\mu_o\prime, \nu\prime, a\prime, b\prime)$, where $\mu_o\prime, \nu\prime, a\prime$ and $b\prime$ are the updated parameters depending on the history of samples $X_{t-k:t-1}$. For the next drawn sample, a way of updating the parameters of the prior $P(\mu|\sigma^2, \mu_o, \nu)P(\sigma^2|a, b)$ is now derived, considering $P(\theta|X) \sim P(X|\theta)P(\theta)$ (see Equation 7.5).

$$
\begin{aligned}
P(\mu, \sigma^2|X_{t-k:t-1}) &= \frac{1}{\sqrt{2\pi\sigma^2\nu}} e^{-\frac{1}{2\sigma^2\nu}(\mu-\mu_o)^2} \frac{b^a}{\Gamma(a)} (\sigma^2)^{-a-1} e^{-\frac{b}{\sigma^2}} \\
P(X_t = x_t, \mu, \sigma^2|X_{t-k:t-1}) &= \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x_t-\mu)^2} \frac{1}{\sqrt{2\pi\sigma^2\nu}} e^{-\frac{1}{2\sigma^2\nu}(\mu-\mu_o)^2} \frac{b^a}{\Gamma(a)} (\sigma^2)^{-a-1} e^{-\frac{b}{\sigma^2}} \\
&= \frac{b^a}{2\pi\sigma^2\nu^{1/2}\Gamma(a)} (\sigma^2)^{-a-1} e^{-\frac{b}{\sigma^2}} e^{-\frac{1}{2\sigma^2}((x_t-\mu)^2+\frac{1}{\nu}(\mu-\mu_o)^2)} \\
&= \frac{b^a}{2\pi\sigma^2\nu^{1/2}\Gamma(a)} (\sigma^2)^{-a-1} e^{-\frac{b}{\sigma^2}} e^{-\frac{(\nu+1)}{2\sigma^2\nu}(\mu-\frac{(\nu x_t+\mu_o)}{(\nu+1)})^2} e^{-\frac{1}{2\sigma^2\nu}(x_t-\mu_o)^2} \\
&= \frac{b^a}{2\pi\nu^{1/2}\Gamma(a)} (\sigma^2)^{-(a+1)-1} e^{-\frac{1}{\sigma^2}(b+\frac{1}{2(\nu+1)}(x_t-\mu_o)^2)} e^{-\frac{(\nu+1)}{2\sigma^2\nu}(\mu-\frac{(\nu x_t+\mu_o)}{(\nu+1)})^2}
\end{aligned}
\tag{7.8}
$$

An expression for the updating of the parameters has been found:

$$\mu_o\prime = \frac{(\nu x_t + \mu_o)}{(\nu + 1)}$$

$$\nu\prime = \frac{\nu}{\nu + 1}$$

$$a\prime = a + 1$$

$$b\prime = b + \frac{1}{2(\nu + 1)}(x_t - \mu_o)^2$$

The expression in Equation 7.8 does not look exactly like a NIG because it is not normalized, but by doing:

$$P(\mu, \sigma^2 | X_{t-k:t-1}, X_t = x_t) = \frac{P(X_t = x_t, \mu, \sigma^2 | X_{t-k:t-1})}{\int_{\mu,\sigma^2} P(X_t = x_t, \mu, \sigma^2 | X_{t-k:t-1})}$$

the expression of a NIG distribution can be found.

A way of updating the parameters μ_o, ν, a and b was found, but no mention was done to how to obtain their values depending on $X_{t-k:t-1}$. Now, returning to Equation 7.1:

$$\int_{\mu,\sigma^2} P(X_t, \mu, \sigma^2 | X_{t-k:t-1}) = P(X_t | X_{t-k:t-1}) \tag{7.9}$$

And considering k to be the run length at $t-1$, this relationship can be found:

$$P(X_t | X_{t-k:t-1}) = P(X_t | X_{t-1}^{(r=k)}) = P(X_t | r_{t-1} = k, X_{1:t-1}) \tag{7.10}$$

Equation 7.10 is exactly what one was originally looking for in 7.4. In order to obtain an expression for it, the integral in Equation 7.9 is calculated.

$$\int_0^\infty \int_{-\infty}^\infty \frac{b^a}{2\pi\nu^{1/2}\Gamma(a)}(\sigma^2)^{-(a+1)-1}e^{-\frac{1}{\sigma^2}(b+\frac{1}{2(\nu+1)}(x_t-\mu_o)^2)}e^{-\frac{(\nu+1)}{2\sigma^2\nu}(\mu-\frac{(\nu x_t+\mu_o)}{(\nu+1)})^2}d\mu d\sigma^2 =$$
$$\int_0^\infty \frac{b^a}{2\pi\nu^{1/2}\Gamma(a)}(\sigma^2)^{-(a+1)-1}e^{-\frac{1}{\sigma^2}(b+\frac{1}{2(\nu+1)}(x_t-\mu_o)^2)}\left(\int_{-\infty}^\infty e^{-\frac{(\nu+1)}{2\sigma^2\nu}(\mu-\frac{(\nu x_t+\mu_o)}{(\nu+1)})^2}d\mu\right)d\sigma^2 =$$
$$\int_0^\infty \frac{b^a}{2\pi\nu^{1/2}\Gamma(a)}(\sigma^2)^{-(a+1)-1}e^{-\frac{1}{\sigma^2}(b+\frac{1}{2(\nu+1)}(x_t-\mu_o)^2)}\left(\int_{-\infty}^\infty e^{-\frac{(\nu+1)}{2\sigma^2\nu}\mu^2}d\mu\right)d\sigma^2 =$$
$$\int_0^\infty \frac{b^a}{2\pi\nu^{1/2}\Gamma(a)}(\sigma^2)^{-(a+1)-1}e^{-\frac{1}{\sigma^2}(b+\frac{1}{2(\nu+1)}(x_t-\mu_o)^2)}\sqrt{\frac{2\pi\sigma^2\nu}{(\nu+1)}}d\sigma^2 =$$
$$\int_0^\infty \frac{b^a}{(2\pi)^{1/2}(\nu+1)^{1/2}\Gamma(a)}(\sigma^2)^{-(a+1)-1/2}e^{-\frac{1}{\sigma^2}(b+\frac{1}{2(\nu+1)}(x_t-\mu_o)^2)}d\sigma^2$$

The following identity is used to solve the integral:

$$\int_0^\infty \delta x^\alpha e^{-\frac{\beta}{x}}dx = \delta\beta^{\alpha+1}\Gamma(-\alpha-1)$$

Then:

$$P(X_t|r_{t-1}=k, X_{t-1}^{(r=k)}) = P(X_t|X_{t-k:t-1})$$
$$= \frac{\Gamma(a+1/2)}{\Gamma(a)(2\pi b)^{1/2}(\nu+1)^{1/2}}\left(1+\frac{1}{2(\nu+1)b}(X_t-\mu_o)^2\right)^{-(a+\frac{1}{2})} \tag{7.11}$$

Equation 7.11 is the generalized t-Student distribution.

7.2.2 Case of the Normal Prior

The assumption now is that the distribution from where X_t is drawn, is a Gaussian with a fixed variance and a moving mean. Then:

$$\begin{aligned} P(X_t, \mu | X_{t-k}, X_{t-(k-1)} ... X_{t-1}) &= P(X_t, \mu | X_{t-k:t-1}) \\ &= P(X_t | \mu, X_{t-k:t-1}) P(\mu | X_{t-k:t-1}) \end{aligned}$$

Given that $X_t \sim N(\mu, \sigma^2)$ and all the Xs are considered *iid*, X_t depends uniquely on μ and σ^2, then:

$$P(X_t, \mu | X_{t-k}, X_{t-(k-1)} ... X_{t-1}) = P(X_t | \mu) P(\mu | X_{t-k:t-1})$$

Distribution $P(\mu | X_{t-k:t-1})$ is now a normal distribution with mean μ_o and variance σ_o^2. The value of μ_o will be a function of $X_{t-k:t-1}$. Then:

$$\begin{aligned} P(X_t, \mu | \mu_o(X_{t-k:t-1}), \sigma_o(X_{t-k:t-1})) &= P(X_t | \mu) P(\mu | \mu_o(X_{t-k:t-1}), \sigma_o(X_{t-k:t-1})) \\ &= P(X_t | \mu) P(\mu | \mu_o, \sigma_o) \end{aligned} \tag{7.12}$$

Equation 7.12 again hides the dependence of X_t and μ on $X_{t-k:t-1}$ through the parameters μ_o and σ_o. In this case the moving parameters are $\vec{\theta} = (\mu)$, and $P(\vec{\theta})$ is a normal distribution with mean μ_o and standard deviation σ_o. From Bayes in

Equation 7.5:

$$P(\vec{\theta}) = P(\mu) = N(\mu|\mu_o, \sigma_o^2)$$

$$P(X|\vec{\theta}) = P(X_t|\mu) = N(X_t, \mu, \sigma^2)$$

Since in this case $P(X|\vec{\theta})$ is normally distributed and $P(\vec{\theta})$ is also normally distributed, then $P(\vec{\theta}|X)$ will also be a normal, since the normal distribution is the conjugate prior for itself.

For a particular value of X_t $P(X_t = x_t, \mu|X_{t-k:t-1}) \propto P(\mu|X_{t-k:t-1}, X_t = x_t)$, then it can be said that $P(X_t = x_t, \mu|X_{t-k:t-1}) \propto N(\mu_o\prime, \sigma_o\prime)$, where now $\mu_o\prime, \sigma_o\prime$ are the updated parameters. For the next drawn sample, a way of updating the parameters of the prior $P(\mu|\mu_o, \sigma_o)$ is now derived.

$$\begin{aligned}
P(\mu|X_{t-k:t-1}) &= \frac{1}{\sqrt{2\pi\sigma_o^2}} e^{-\frac{1}{2\sigma_o^2}(\mu-\mu_o)^2} \\
P(X_t = x_t, \mu|X_{t-k:t-1}) &= \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(x_t-\mu)^2} \frac{1}{\sqrt{2\pi\sigma_o^2}} e^{-\frac{1}{2\sigma_o^2}(\mu-\mu_o)^2} \\
&= \frac{1}{2\pi\sigma\sigma_o} e^{-\frac{1}{2\sigma^2}(x_t-\mu)^2 - \frac{1}{2\sigma_o^2}(\mu-\mu_o)^2} \\
&= \frac{1}{2\pi\sigma\sigma_o} e^{-\frac{1}{2}(\frac{x_t^2}{\sigma^2} + \frac{\mu^2}{\sigma^2} - \frac{2x_t\mu}{\sigma^2} + \frac{\mu^2}{\sigma_o^2} + \frac{\mu_o^2}{\sigma_o^2} - \frac{2\mu\mu_o}{\sigma_o^2})} \\
&= \frac{1}{2\pi\sigma\sigma_o} e^{-\frac{1}{2}(\frac{1}{\sigma^2} + \frac{1}{\sigma_o^2})(\mu - \frac{(\sigma_o^2 x_t + \mu_o\sigma^2)}{(\sigma_o^2+\sigma^2)})^2} e^{-\frac{1}{2(\sigma^2+\sigma_o^2)}(x-\mu_o)^2} \qquad (7.13)
\end{aligned}$$

An expression to update the parameters can now be formulated:

$$\mu_o\prime = \frac{(\sigma_o^2 x_t + \mu_o \sigma^2)}{(\sigma_o^2 + \sigma^2)}$$
$$\sigma_o^2\prime = \left(\frac{1}{\sigma^2} + \frac{1}{\sigma_o^2}\right)^{-1}$$

The expression in Equation 7.13 does not look exactly like a normal distribution because it is not normalized. Then by doing:

$$P(\mu|X_{t-k:t-1}, X_t = x_t) = \frac{P(X_t = x_t, \mu|X_{t-k:t-1})}{\int_\mu P(X_t = x_t, \mu|X_{t-k:t-1})}$$

the expression of a normal distribution can be found. Now, returning to Equation 7.1:

$$\int_\mu P(X_t, \mu|X_{t-k:t-1}) = P(X_t|X_{t-k:t-1}) \tag{7.14}$$

And considering k to be the run length at $t-1$, then the following relationship is found:

$$P(X_t|X_{t-k:t-1}) = P(X_t|X_{t-1}^{(r=k)}) = P(X_t|r_{t-1} = k, X_{1:t-1}) \tag{7.15}$$

Equation 7.15 is again what one was originally looking for in 7.1. In order to

obtain an expression for it, the integral in Equation 7.14 is now calculated.

$$\int_{-\infty}^{+\infty} \frac{1}{2\pi\sigma\sigma_o} e^{-\frac{1}{2}(\frac{1}{\sigma^2}+\frac{1}{\sigma_o^2})(\mu-\frac{(\sigma_o^2 x_t+\mu_o\sigma^2)}{(\sigma_o^2+\sigma^2)})^2} e^{-\frac{1}{2(\sigma^2+\sigma_o^2)}(x-\mu_o)^2} =$$
$$\frac{1}{2\pi\sigma\sigma_o} e^{-\frac{1}{2(\sigma^2+\sigma_o^2)}(x-\mu_o)^2} \int_{-\infty}^{+\infty} e^{-\frac{1}{2}(\frac{1}{\sigma^2}+\frac{1}{\sigma_o^2})(\mu-\frac{(\sigma_o^2 x_t+\mu_o\sigma^2)}{(\sigma_o^2+\sigma^2)})^2} =$$
$$\frac{1}{2\pi\sigma\sigma_o} e^{-\frac{1}{2(\sigma^2+\sigma_o^2)}(x-\mu_o)^2} \int_{-\infty}^{+\infty} e^{-\frac{1}{2}(\frac{1}{\sigma^2}+\frac{1}{\sigma_o^2})\mu^2} =$$
$$\frac{1}{2\pi\sigma\sigma_o} e^{-\frac{1}{2(\sigma^2+\sigma_o^2)}(x-\mu_o)^2} \sqrt{2\pi}\sqrt{\frac{\sigma^2\sigma_o^2}{\sigma^2+\sigma_o^2}} =$$
$$\frac{1}{\sqrt{2\pi(\sigma^2+\sigma_o^2)}} e^{-\frac{1}{2(\sigma^2+\sigma_o^2)}(x-\mu_o)^2}$$

It can now be said that:

$$P(X_t|r_{t-1}=k, X_{t-1}^{(r=k)}) = P(X_t|X_{t-k:t-1}) = \frac{1}{\sqrt{2\pi(\sigma^2+\sigma_o^2)}} e^{-\frac{1}{2(\sigma^2+\sigma_o^2)}(x-\mu_o)^2} \quad (7.16)$$

7.2.3 Step by Step Algorithm Computation

All the different terms in Equation 7.1 have been defined. The steps taken to perform this algorithm in an online fashion are presented. These steps can be found in,[48] but here a more clear explanation is provided.

1. **Initialize**. $r_{(t=0)}$ can only take one value, 0, and then $P(r_{t=0} = 0) = 1$. If somehow information about the previous state of the process is available, then distribution $P(r_{t=0})$ can start off that provided distribution. The case used here is the one in which nothing is known about the process for time before $t = 0$.

At every time step, r will have the possibility of being different values, and since the whole last subsection was developed for a particular value of $r = k$, then for every value of r a different set of values for μ_o, ν, a and b for the case of the gamma inverse prior, and μ_o and σ_o for the case of the normal prior will be found for every time step. Variable $\vec{\theta}$ will be considered to be μ_o, ν, a and b for the case of the gamma inverse prior, and μ_o and σ_o for the normal prior. The initial values for $\vec{\theta}$ are set to $\vec{\theta_o}$, which are considered to be the best guess for when nothing is known.

2. **Observe New Datum x_t.**

3. **Evaluate the predictive probability.**

$$P(X_t|r_{t-1}, X_{t-1}^{(r)} = x_{t-1}^{(r)}) = \pi_t^{(r)}$$

4. **Calculate Growth Probabilities.**

$$P(r_t = r_{t-1} + 1, X_{1:t} = x_{1:t}) = P(r_{t-1}, X_{1:t-1} = x_{1:t-1})\pi_t^{(r)}(1 - H(r_{t-1})) \quad (7.17)$$

5. **Calculate Change-Point Probabilities.** In this step, if the probability is considered to be high enough, it can be considered that a Change-Point has

been found.

$$P(r_t = 0, X_{1:t} = x_{1:t}) = \sum_{r_{t-1}} P(r_{t-1}, X_{1:t-1} = x_{1:t-1}) \pi_t^{(r)} H(r_{t-1}) \tag{7.18}$$

6. **Calculate evidence.**

$$P(X_{1:t} = x_{1:t}) = \sum_{r_t} P(r_t, X_{1:t} = x_{1:t}) \tag{7.19}$$

7. **Determine Run Length Distribution.**

$$P(r_t | X_{1:t} = x_{1:t}) = P(r_t, X_{1:t} = x_{1:t}) / P(X_{1:t} = x_{1:t}) \tag{7.20}$$

8. **Update the parameters.** In understanding this step, an example is given. At time $t = 1$, r can be either 0 or 1, so two are the number of sets of parameters. The second set of parameters, for which $r = 1$, will be updated using the parameters corresponding to the previous time step for $r = 0$. This happens for all the values of r, except $r = 0$, for which one starts with the parameters from step 1.

$$(\vec{\theta})_t^{r+1} = f(X_t = x_t, (\vec{\theta})_{(t-1)}^r) \tag{7.21}$$

9. **Return to step 2.**

7.3 Stochastic Computing

7.3.1 Introduction

A detailed mathematical explanation of the ChangePoint Detection algorithm was presented. Two options were considered for the prior distribution $P(\vec{\theta}|X_{1:t-1})$, where the first one was the Inverse Gamma distribution, and the second one was a Normal distribution. The first one considers both variance and mean to be changing over time, and the second one considers only the mean to change. The first option would result in $P(X_t|r_{t-1} = k, X_{1:t-1})$ being a t-Student distribution, which needs to be evaluated for every new drawn sample. On the other hand, the second approach involves evaluating a simpler Normal distribution. The CPD algorithm is very convenient from a parallel computation point of view, as each of the pixels in an image can be evaluated with this algorithm completely independent from any other pixel. Because of the large size of the images processed by the CMPs, the second approach was the one considered for this project.

The program funding this project was *UPSIDE* from *DARPA*, which stands for *Unconventional Processing of Signals for Intelligent Data Exploitation*. One of the main objectives in this program was the research of new unconventional ways of performing computations. It is for this reason that stochastic computing was considered, as it allowed in its compact representation of signals as one-bit streams to achieve digital architectures much smaller than the ones that would result from the conven-

tional use of binary coded values. With the reduction of silicon area, a decrease in the operating power would also be seen. A careful review of some of the basic stochastic operational units were studied in.[56–60] These elements would allow, for instance, the calculation of multiplications by using only an AND gate, or the calculation of a low precision division by using a simple JK flip-flop. Further architectures using these type of computational units can be seen in.[61–63]

One very important aspect to take into account when considering stochastic computing, is that the representation domain for these operations is not the binary coded one. The domain of operation for these computing elements is inherently stochastic, meaning that the numbers used in the calculations are probability values. Every number is represented as a stationary random process, where each time step can be assigned either a '0' or a '1', making each of these time steps Bernoulli distributed. Conventional computing uses numbers $x \in \mathbb{R}$ for the domain of the input and output arguments in a computational unit. On the other hand, for the case of stochastic computing, input and output arguments are bounded by $\mathbb{R} \in [0; 1]$. In using this representation, one needs to apply a linear transformation $\mathscr{T} : \mathbb{R} \Rightarrow \mathbb{R} \in [0; 1]$, for every single input argument. Let's consider the case of a simple multiplication, were both multiplicand and multiplier are two 4-bit numbers 4 and 9. Because the maximum number represented in four bits is 15, 16 will be mapped to a Bernoulli probability $p = 1$, and the minimum represented number 0 will be encoded with $p = 0$. When now 4 and 9 need to be represented in probabilities, one can obtain these probabilities by

doing $p_4 = 4/16$ and $p_9 = 9/16$. When looking at an AND gate, one can say that the probability of '1' at the output of that gate is the probability of both inputs being '1', which is the multiplication of probabilities. Now if two stationary Bernoulli processes can be generated with probabilities 4/16 and 9/16, then the output of the AND gate will be a stationary Bernoulli process with probability $p = (4/15)(9/15) = 36/256$. Because each of the inputs is represented as a 4-bit number, the multiplication will generate an 8-bit number, and then if one wants to convert the output probability to conventional binary representation, a multiplication by 256 needs to take place.

When dealing with stochastic computing two major transformations are involved. The first transformation is the translation of a probability value into a random stream of zeros and ones. This first transformation is what it's usually called *Encoding* transformation. An example of an encoder is presented in Figure 7.2. This encoder requires the usage of a uniform random number source and a comparator to perform the stochastic encoding.

The second transformation is the *Decoding* of a stochastic representation of numbers into binary coded. This translation can be easily done by using an estimator of the represented Bernoulli probability p. The estimator is the well-known mean estimator:

$$\hat{p} = \frac{1}{N} \sum_{t=1}^{N} X(t) \tag{7.22}$$

where $X(t)$ are samples from the stochastic stream one wants to decode. A very important thing needs to be kept in mind for the decoding transformation, and that

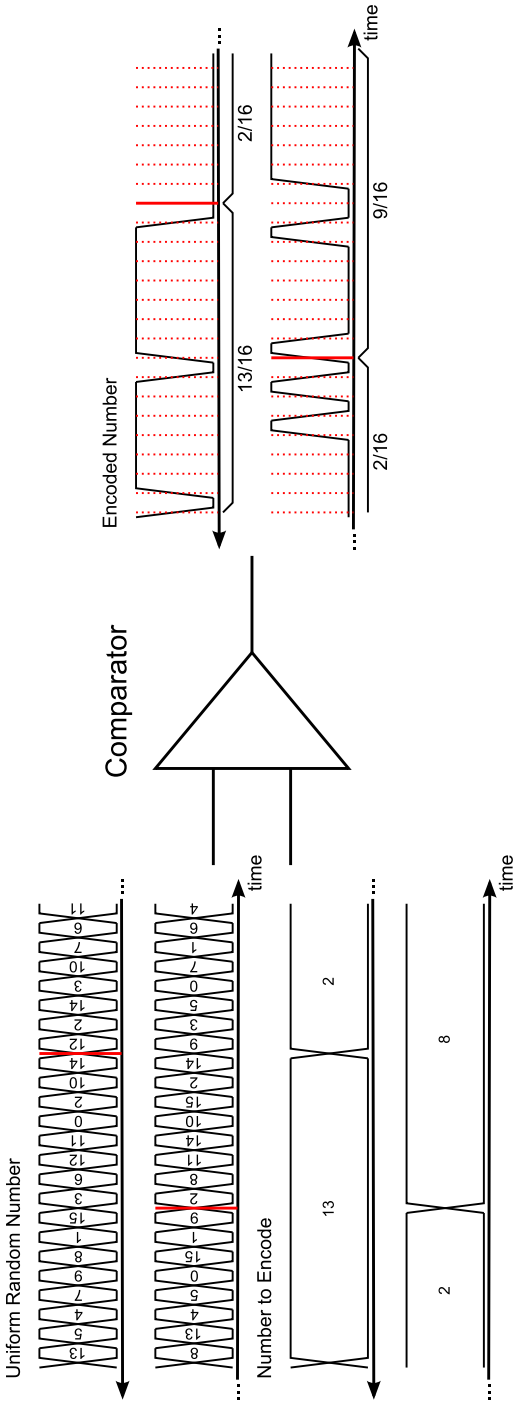


Figure 7.2: Stochastic Encoder. This example shows the change of the encoded number every 16 time slots. A 4-bit uniform random number generator is used in the encoding. At the output of the comparator, for the encoded values 13, 2 and 8, 13, 2 and 9 were the number of '1's found on each number period. Number 8 does not translate into eight '1's because the numbers used for the encoding are random.

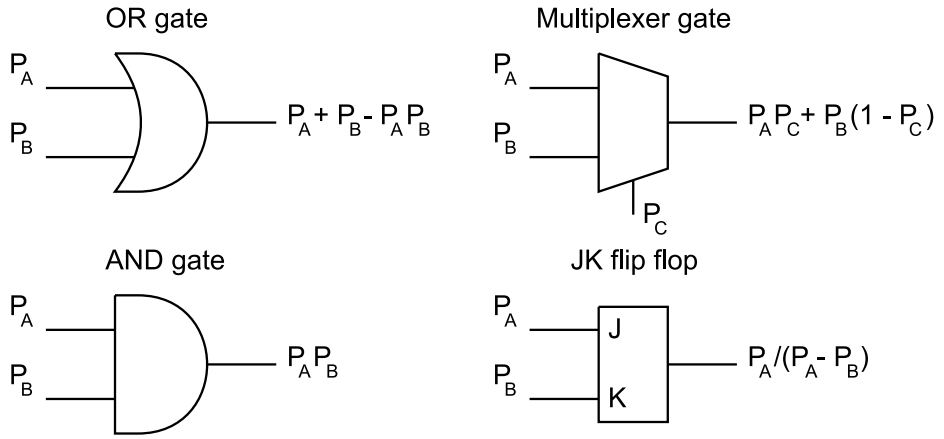


Figure 7.3: Stochastic computation elements. Example of four stochastic computational elements, the OR gate, AND gate, two-input multiplexer gate and JK flip flop. At the inputs and output of these gates the probability conversion is presented.

it is done through the usage of an estimator. Any kind of estimation is done with certain accuracy, meaning that errors are inherent to the estimation. This means that the decoding mechanism is not a loss-less transformation, and then it is recommended to be performed in a stochastic processor as few times as possible. For $N > 30$, using the *Central limit theorem* the distribution of estimator $\hat{p}$ is approximated as Gaussian and can be expressed as:

$$\hat{p} \sim \mathcal{N}(\mu = p, \sigma = \sqrt{N}) \tag{7.23}$$

The estimation error can be calculated with certain confidence using the expression in Equation 7.23. The effect of this error can be seen in Figure 7.2, where the number 8 can be estimated at the output of the comparator for $N = 16$ as 9. Figure 7.3 presents four different stochastic elements used when building stochastic machines.

As one can observe, due to the decoding error found in Equation 7.23, it is to expect that stochastic computation should not be performed when high accuracy is

needed. The increase of the time needed to perform the decoding increases quadratically with a linear decrease of the error in the estimation. The advantage of this stochastic approach to computations is that not only the silicon area required to perform very complicated computational operations such as a multiplication or division are reduced dramatically, but it also allows to perform accuracy on demand. Depending on the task at hand, more or less time can be used in the decoding of signals, allowing power to be more efficiently managed.

7.3.2 Stochastic Architecture for the Online CPD Algorithm

In order to perform background-foreground segmentation on images, a stochastic architecture was built for the CPD Equation 7.1, using the recurring steps presented in 7.2.3. In the implementation presented in this work, from Equation 7.2, $H(r_{t-1}+1) = 1/\lambda$. Considering the case of the normal prior, the update of parameters becomes:

$$\mu_{ok}\prime = \frac{\sigma^2_{o(k-1)}x_t + \mu_{o(k-1)}\sigma^2}{\sigma^2_{o(k-1)}+\sigma^2} \quad \sigma^2_{ok}\prime = \left(\frac{1}{\sigma^2} + \frac{1}{\sigma^2_{o(k-1)}}\right)^{-1} \tag{7.24}$$

The value k represents a possible run-length value. The parameters for run-length k will then depend on the current received sample, and the parameters for run-length $k-1$.

In contrast with the original CPD formulation, the proposed CPD implementation

will have the run-length distribution trimmed, meaning that run-lengths higher than $Nwin - 1$ will not be considered for the run-length distribution $P(r_t|X_{1:t})$. If no information of the time before the algorithm starts is provided, then the default distribution for $P(r_t|X_{1:t})$ will have all of its weight in $P(r_0 = 0|X_{1:t}) = 1$. At time $t =$ *(Nwin-1)*, distribution $P(r_t|X_{1:t})$ will have been populated with values different than zero for run-lengths up to $Nwin - 1$, and parameters $\vec{\phi}_k$ $k \in \{0, 1, 2, ..., Nwin - 1\}$ corresponding to run-lengths $r = k$ will have already been generated. At this point, distribution $P(r_t|X_{1:t})$ can then be already evaluated for *Nwin* different values (*0* to *Nwin-1*). Since up to *Nwin* values are stored for the run-length distribution $P(r_t|X_{1:t})$, the moment the following $X_{t=Nwin}$ sample arrives, the parameters $\vec{\phi}_k = (\sigma^2_{ok}, \mu_{ok})$ for $k =$ *Nwin* will not be generated, and the probability value assigned for $r_{Nwin} =$ *Nwin*, will be added to the probability calculated for $r_{Nwin} =$ *Nwin-1*. This way r_t is always limited to a constant number of *Nwin* values over time. Furthermore, the value $P(r_t =$ *Nwin-1*$|\vec{\phi}_{(Nwin-1)}) + P(r_t =$ *Nwin*$|\vec{\phi}_{(Nwin)})$ will have to be assigned to $P(r_t =$ *Nwin-1*$|\vec{\phi}_{(Nwin-1)})$. By doing this, $P(r_t =$ *Nwin-1*$|\vec{\phi}_{(Nwin-1)})$ can be intepreted as the probability of Change-Point not only for $r_t = Nwin - 1$ but for all $r_t \geq Nwin - 1$.

A diagram of the designed architecture for the CPD algorithm is presented in Figure 7.4. At time t, registers $Ro(0)$ to $Ro(Nwin-1)$ found in Figure 7.4 (1) will hold the probability distribution values for $P(r_{t-1}|X_{1:t-1})$. These registers simply contain count values (integer values), and they do not necessarily represent the normalized

version of $P(r_{t-1}|X_{1:t-1})$. Register Wo will contain the sum of all the values of the $\vec{Ro}$ registers, so this can be thought of as the normalizing constant.

When a new sample value arrives, all of the values in the $\vec{Ro}$ registers need to be encoded stochastically, so several comparators and random number generators are necessary to generate the stochastic streams (see Figure 7.4 (2)). In order to perform a normalization on the stochastically encoded $\vec{Ro}$ values, the statistical characteristics at the output of a JK flipflop will help do that (see Figure 7.3). By using stochastic adders/subtractors, binary comparators and JK flip flops, the sum of all the Bernoulli probabilities at the output of the JK flip flops will add to one, making the normalization of $P(r_{t-1}|X_{1:t-1})$ possible without involving complicated division algorithms. The stochastic adder/subtractor mentioned is the one represented by a circle in Figure 7.4 (2). This unit will contain a counter that will increase its count by the difference of its inputs, where that difference can be -1, 0 or 1. After adding that step to the local counter, if the local counter holds a value greater than 0, then a '1' is forwarded to the output, and the counter is decreased by a count.

The length of these stochastic streams can be changed depending on the accuracy required from the algorithm. If a higher accuracy is required, more computational time has to be provided. On the right side of registers $\vec{Ro}$, registers $Ri(0)$ to $Ri(Nwin\text{-}1)$ and Wi will contain the updated probability distribution $P(r_t|X_{1:t})$ by the end of the chosen computational time. Every time a new sample X_t arrives, registers $\vec{Ro}$ and Wo are loaded with $P(r_{t-1}|X_{1:t-1})$ held by registers $\vec{Ri}$ and Wi.

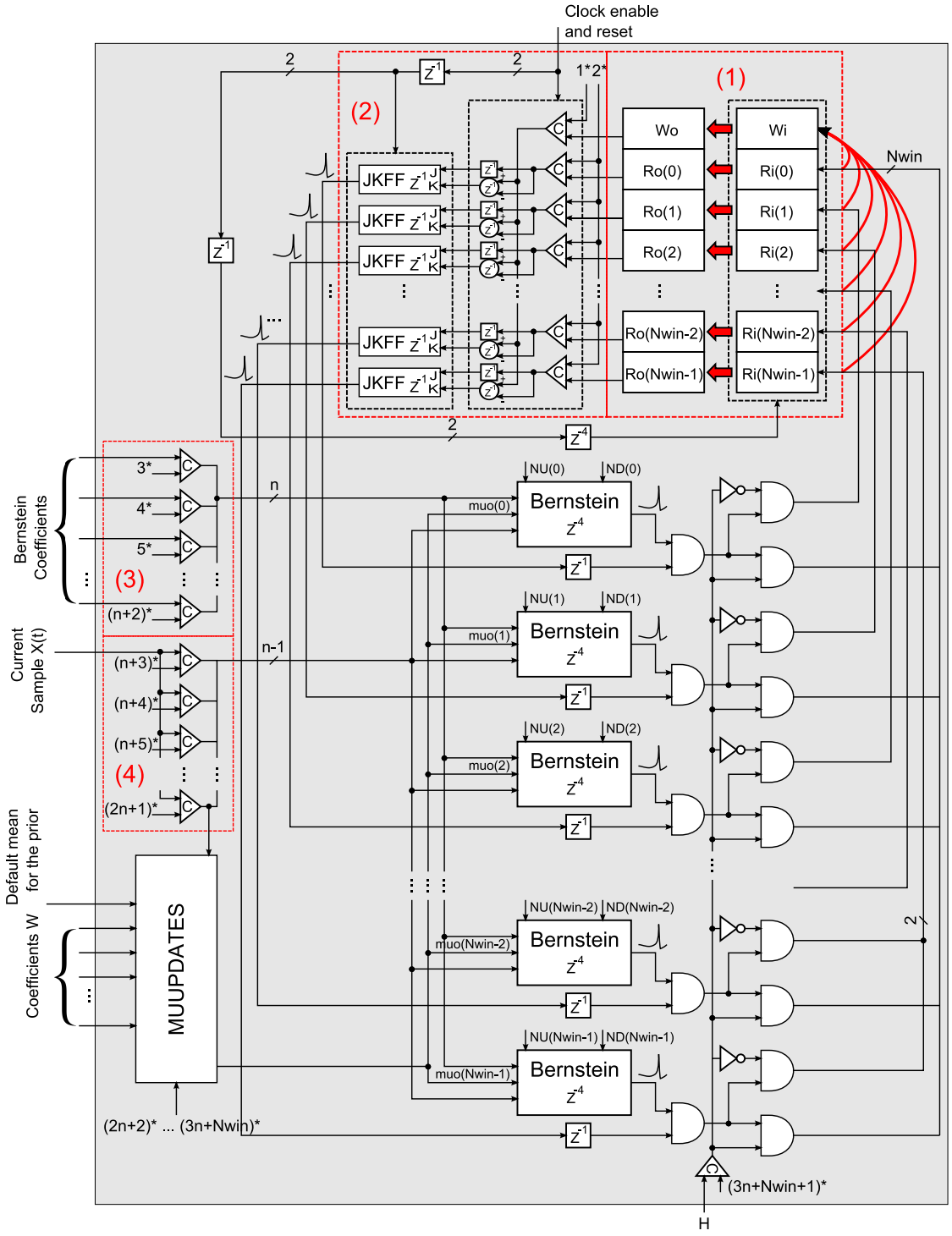


Figure 7.4: Stochastic architecture for the CPD algorithm. All the values with a star represent the independent uniform random number streams required.

After this transference, $\vec{R}i$ and Wi registers are set to zero. Registers $\vec{R}i$ and Wi will integrate the ones from their stochastic inputs, acting as stochastic decoders. After the computation time chosen, the resulting count values in registers $\vec{R}i$ will represent the non-necessarily normalized version of the $P(r_t|X_{1:t})$ distribution. The resulting distribution will not be necessarily normalized because of the nature of this type of stochastic computation. The decoding process has a mean and a variance, and it is because of that variance that the distribution might not add to one.

With a similar structure, the parameters from Equation 7.24 are updated. In fact, notice that the expression for $\sigma^2_{ok}{}' \; \forall k$ does not depend on the value of the current X_t sample, and as a result, they are constants. That makes it possible not having memory states for these parameters. In Figure 7.5 the *Coefficients W* will provide the $\sigma^2_{o(k-1)}/(\sigma^2_{o(k-1)} + \sigma^2)$ constant values, and after being stochastically encoded, will be used in the update of the mean parameters μ_{ok} for $k \in [1 : Nwin - 1]$. The default mean for the prior does not change, and then it is added as one of the inputs in Figure 7.5. The value for this default mean will generally be set to 0.5 for the most uninformed guess. One can estimate the mean of the analyzed random process over a long period of time, and add that estimation as the default mean. Input X_t will be encoded stochastically and will be used with two input multiplexers to perform the necessary weighted sum in the update of the mean parameters (see Equation 7.24). The way in which registers $\vec{m}_{uo}$ and $\vec{m}_{ui}$ are reset and loaded is similar to the $\vec{R}o$ and $\vec{R}i$ registers. It can be also observed in 7.5 how the arrows from the multiplexers

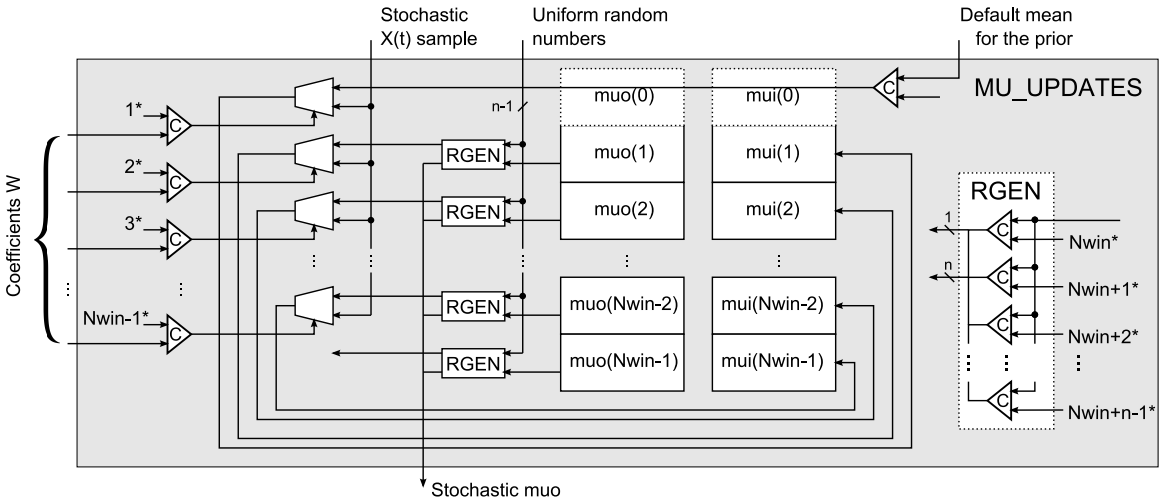


Figure 7.5: Stochastic update of the mean parameters. The means are updated with a weighted sum performed by the two-input multiplexers.

to the $\vec{m}_{ui}$ registers go down one level, meaning that $\mu_{ok}\prime$ depends on $\mu_{o(k-1)}$, as expected. Each of the $\vec{m}_{uo}$ register values is encoded in *n-1* different independent streams. One of those streams is used for the updates in the μ_{ok} parameters, and at the same time all of those streams are sent out to the blocks that will compute the Gaussian distributions $P(X_t|r_{t-1}, X_{t-1}^{(r_{t-1})})$ for the different values of r_{t-1}.

Going back to Figure 7.4 three different types of stochastic streams can be seen, the ones for the encoded X_t value (4), the ones corresponding to the mean parameters μ_{ok} coming out of the *MU_UPDATES* block, and the Bernstein coefficients streams (3) that will be used to generate the required Gaussian function $P(X_t|\vec{\phi}_k)$. The Bernstein polynomials[64] are polynomials that can be used to approximate any function $f(x) = y$ with $x \in [0;1]$ and $y \in [0;1]$, and they rely on the weigthed sum of Bernoulli probabilities. Considering n independent stochastic streams representing the number p, if at every time step all of the ones from the different stochastic streams are added,

the probability of having q as the output addition is a Binomial distribution:

$$P(q) = \binom{q}{n} p^q (1-p)^{n-q} \tag{7.25}$$

By computing a weigthed sum of these probabilities one can approximate the desired function:

$$f(p) \approx \sum_{i=1}^{n} w_i \binom{q}{n} p^q (1-p)^{n-q} = \hat{f}(p) \tag{7.26}$$

In order to make this approximation stochastically feasible, one needs the weights w_i to be values $\in [0; 1]$. The architecture for this Bernstein polynomial function approximation can be found in Figure 7.6. By chosing among the different Bernstein coefficient streams with a multiplexer, the desired weighted effect is obtained. The distribution $P(X_t|\vec{\phi}_k)$ is Gaussian, and its parameters are $(\mu_k = \mu_{ok}, \sigma_k^2 = \sigma^2 + \sigma_{ok}^2)$. The block ABS performs the stochastic absolute value of the difference between the inputs, so that only half of the Gaussian bell needs to be approximated by the Bernstein polynomials. The problem that now arises is that $Nwin$ different Gaussians with $Nwin$ possibly different variances need to be approximated. This would mean that $Nwin$ different sets of Bernstein coefficients are needed, incrementing significantly the number of coefficients supplied to the architecture that need to be stochastically encoded. As a solution to this problem, the concept of a bursting neuron was applied. If a Gaussian distribution with standard deviation 0.2 was approximated with certain coefficients, how can these same coefficients be used to approximate a Gaussian with

standard deviation $0.2j/i$ with $i, j \in \mathbb{N}$? Some neurons when excited, they generate a train of impulses at their output instead of a single spike. This same idea will be applied to the *BURST UP* blocks in Figure 7.6. For every spike at the input, $NU(*)$ spikes will be generated at the output. If the original Gaussian has a standard deviation of 0.2, and i spikes are generated at the output of these blocks, then the approximated half Gaussian will decrease its standard deviation to $0.2/i$. On the other hand for blocks *BURST DOWN*, if only every j spikes, one spike is generated at the output, then the effect is the opposite. By concatenating these two blocks, a more accurate approximation for the standard deviations in 7.24 can be obtained, as two different degrees or freedom can be used in such approximation.

In the scaling of the streams going to the Bernstein approximation blocks there is a caveat. Even if it seems that the use of two consecutive blocks that multiply and divide by constants stochastically can help to achieve a better accuracy for the required standard deviations in Equation 7.24, there is a problem involved. For the case of the *Burstdown* block, a spike is sent to the output every time j spikes were received at the input, and for *Burstup* block, i spikes are sent at the output every time a spike is received at the input. When processing a new sample X_t, during the processing time N, K ones may be received at the input of the *Bernstein* blocks, meaning that the signal value is approximately K/N. When the division is performed on this value, a remainder *rem* could be left in the calculation. This value is $< j/N$, and when scaling by i, a maximum accuracy error of $(j-1)i/N$ is suffered. This

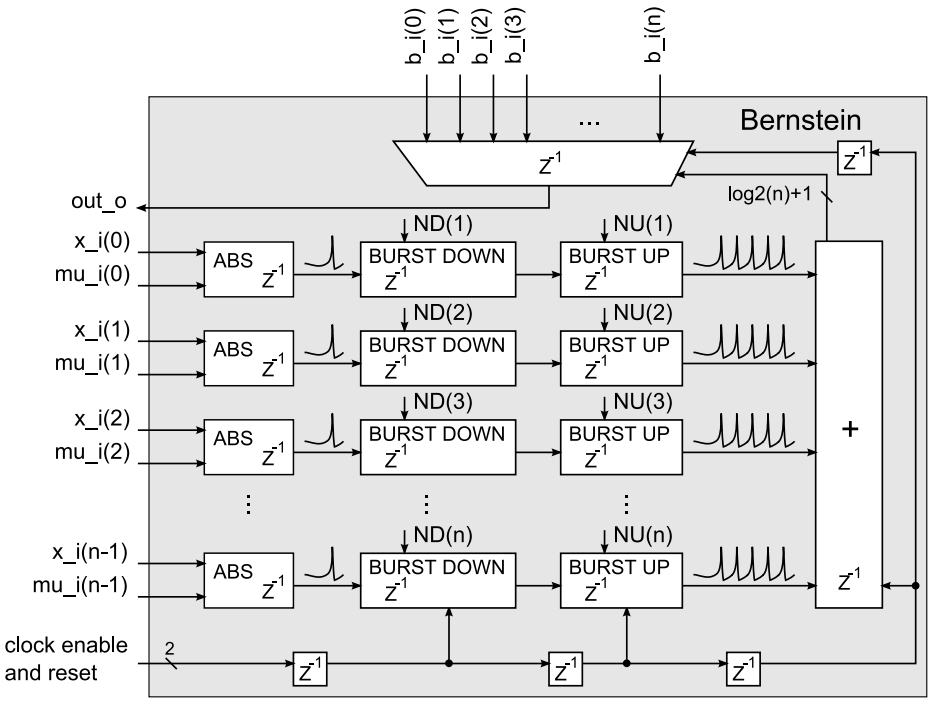


Figure 7.6: Bernstein polynomials block. Architecture used in the approximation of a half Gaussian bell.

error wouldn't exist if *Burstdown* block was disabled by setting $j = 1$, but a lower accuracy would be obtained for the standard deviations in 7.24. Additionally, when encoding small probability values, the relative error could become high. Overall, simulations performed showed that even in this case, the behavior of the system improved compared to the disabling of the *Burst down* blocks.

The combinatorial circuit placed after the *Bernstein* blocks can be easily explained considering that the *AND* gate performs the multiplication operation, and the *NOT* gate performs $1 - p$, where p is the value encoded at the input. To understand the reason of all of the connections that go to the input of the counters $\vec{R}i$ and Wi one can go back to equations 7.17 and 7.18.

The total number of uncorrelated uniform random numbers used in this design is $3n + Nwin + 1$. Many of these numbers are used more than one time because the generated stochastic streams are not crossing their paths. If they do cross their paths it is only at the input of the $\vec{Ri}$ and Wi registers, which is not a problem because there are no more stochastic computations performed for which correlation could be a problem.

7.4 Stochastic CPD Test Chips GF1 & GF2

Two chips were fabricated implementing the presented CPD stochastic architecture. Both chips are functionally identical, but the first one (GF1) was implemented using the original *IBM* standard cell library in $65nm$ GF process, and the second one (GF2) was implemented using the new redesigned standard cell library mentioned in Chapter 6. For the case of the second design, this one was fabricated in $55nm$ GF process. Any gds designed in $65nm$, for the case of *Global Foundries*, can be used for the $55nm$ process, because a 10% optical shrink is performed on $65nm$ designs to convert them into $55nm$ process compatible.

In these fabricated chips, a programmable version of the stochastic CPD was implemented. Four of the structures in Figure 7.4 were put together, using a basic serial interface to program all of the parameters on-chip. The programmability of all

of the cores on-chip is limited as all of the cores have to be programmed in the same way, meaning that all of their parameters have to be equal. This signifies that the four different analyzed signals by the four cores will have to be of the same *nature*, meaning, for instance, that noise and signal strengths will have to be similar.

In the previous FPGA synthesized versions of the CPD core, computational time was not programmable. For the case of the presented chips, this computational time can actually be programmed, not only allowing precision on demand but also to control power dissipation through that precision. The random number generators used in these chips are programmable LFSRs, meaning that they can change their period depending on the computational time required. An alternative could have been to just have a single free-running LFSR for each of the programmable LFSR, and taking the random numbers from it, even when the computation time required is less than its period. The problem with this scenario is that uniformity of the samples taken over the computational time cannot be ensured, making programmable LFSRs a better option. The LFSR size can be changed from 3 to 20 bits, achieving maximum length for all cases. Also, a maximum size for the time window $Nwin$ was set to 16, as well as the maximum number of coefficients that can be used to approximate half of the Gaussian bell with the Bernstein approximation, which was set to 8. The versatility in programming these CPD cores allow to perform tests, where one can find the optimum numbers for parameters like the time window, or the computational time. This would allow to achieve non-programmable, but much

more compact designs for specific applications.

Figure 7.7 shows the chips' layout and pinout. A simple protocol is used for writing data to the chips, consisting of mainly five signals, an input clock, an enable signal, the serial data input, a reset input and an acknowledge output. First a pulse is sent through the reset input to reset an internal counter. After this, the enable signal will determine when the serial data input is valid, and after a whole word was received, the chip will acknowledge by sending a pulse through the acknowledge output. The chip is aware of the length of the words being written, so it will know when to acknowledge.

The left bank of pins is used to load the parameters into the CPD cores. In this bank, the signals involved in this process are *par_ack_o*, *par_i*, *par_en_i*, *par_reset_i*, and *par_type_i*, which is 4 bits. This last input identifies which parameter is being written. There are nine different parameters and they are briefly explained in table 7.1.

On the top bank, signals required to write the current sample X_t to the CPD cores are found. These signals are *x_reset_i*, *x_en_i*, *x_i* and *x_ack_o*. On the bottom bank signals necessary to write the $m\vec{u}o$ registers in each of the CPD cores are found. These signals are *muo_reset_i*, *muo_en_i*, *muo_i* and *muo_ack_o*. On the right bank signals responsible to write the Wo and $\vec{R}o$ registers can be found. These signals are *woro_reset_i*, *woro_en_i*, *woro_i* and *woro_ack_o*. All of the interfaces mentioned use the same clock input *clk_i*.

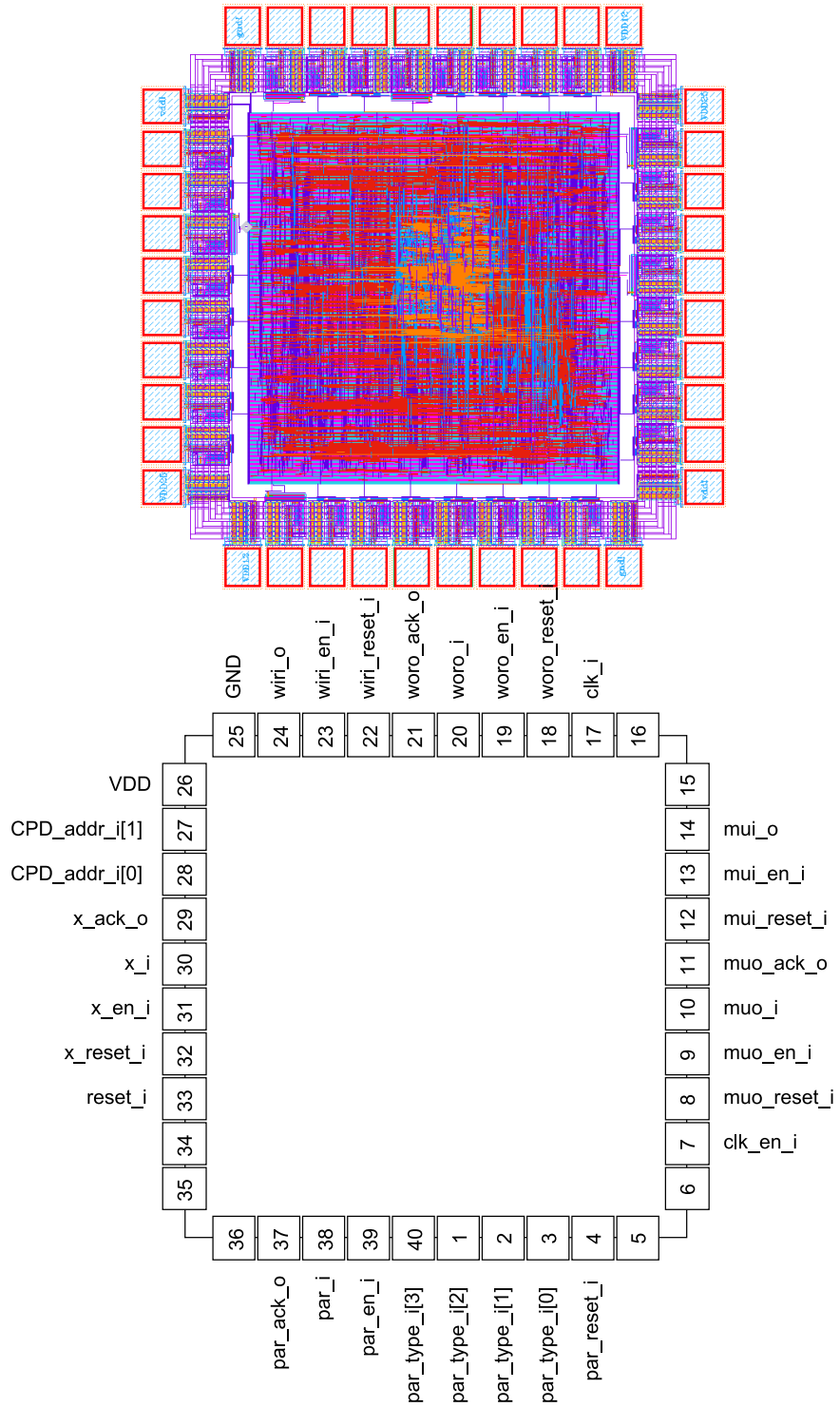


Figure 7.7: GF1 & GF2 layout and pinout. Only one layout is presented as both of them are very similar. On the bottom the pinout and on the top the layout.

Parameters	# of words	# of bits
(type 0) Seeds for the LFSRs	40	20
(type 1) Length of the LFSRs (from 3 to 20)	1	5
(type 2) CPD time window size (from 1 to 16)	1	4
(type 3) Number of Bernstein coefficients (from 1 to 8)	1	3
(type 4) Weight values used in recalculating the means	15	20
(type 5) H value	1	20
(type 6) Bernstein coefficients	8	20
(type 7) Nburst coefficients Up	16	5
(type 8) Nburst coefficients Down	16	5

Table 7.1: Parameters in the GF1 & GF2 test chips. List of the different parameters with the expected number of bits for each one.

At the end of each computational cycle the values from the registers mui, Wi and $\vec{Ri}$ need to be extracted. The extraction of Wi and $\vec{Ri}$ allow to see the evolution of the run-length probability distribution. In reading the mui registers, signals mui_reset_i, mui_en_i and mui_o from the bottom bank of pins are used. For the read of registers Wi and $\vec{Ri}$ signals $wiri\_o$, $wiri\_en\_i$ and $wiri\_reset\_i$ from the right bank of pins are used. Another two-bit signal input CPD_addr_i is provided to address the CPD core of interest from which one desires to read. The way in which the reading protocol works is very simple, a reset pulse is first sent to the chip, and, with a one clock cycle delay and using the enable signal, the registers' values can be read out serially.

In addition to all of the previously mentioned signals, four more signals need to be addressed. At the top of the chip, VDD and GND are the power supply and ground pins for the chip. For the case of the power supply, its nominal value is 1.2V, but because of the usage of low voltage standard cells, operation down to 400mV was

possible for GF2. The last two signals to mention are $reset\_i$ and clk_en_i. Before starting the computation of the next cycle a pulse should be sent through $reset\_i$. When the next sample value X_t is already loaded, as well as registers $m\vec{u}o$, Wo and $\vec{Ro}$, signal clk_en_i needs to be asserted for a number of clock cycles corresponding to the ones specified by parameter *type 1* (see Table 7.1).

The chips tested were mounted on a custom designed board that communicates with an OpalKelly board featuring a Xilinx Spartan3 FPGA. The communication between the FPGA and the chip was done through two high-dense connectors. All the signals driving the chip were provided by the FPGA, even the clock signal, giving the versatility of programming the chip clock frequency very easily from the computer.

The pads for both GF1 and GF2 chips were custom designed, and due to problems in the design of the output pads, only frequencies lower than 12Mhz were functional. Unfortunately, GF2 chip was submitted before these issues were fixed. In spite of this problem, both chips are functional.

A *Matlab* program and *VHDL* code was written for testing the chip. This program, depending on the statistical parameters of the signals to be analyzed, calculates all the parameters for the CPD chip. Those parameters are translated into values that are sent to the chip serially. Both GF1 and GF2 were tested successfully. The same sequence of samples was sent to all of the cores simultaneously so that the same behavior (or similar behavior, since computations are done stochastically) is confirmed. Figure 7.8 shows the results extracted from the chip using $Nwin = 8$ and

a computational time of 2048 clock cycles. Two of the cores worked just perfectly, showing results almost equal to the ones found in the non-stochastic implementation of the algorithm in *Matlab*. The other two cores did not work exactly the way expected. The data received from the chip showed that the sent samples are being taken by those cores scaled down, meaning that the samples received by the cores were interpreted as divided by a constant. This behavior seen in cores 2 and 3, even if it is undesired, shows that they are still creating an output probability density function similar to the ones the fully functional cores generate.

The way in which ChangePoints are found in Figure 7.8 will now be explained. The run-length distribution $P(r_t|X_{1:t})$ is the one used to decide when a change is considered to have happened. If at time $t = 0$ a ChangePoint has been found, triggering an alarm, then a new alarm will be triggered in the subsequent time steps if:

$$\sum_{k=0}^{\gamma} P(r_t = k|X_{1:t}) > threshold, \gamma = min(t - t_{alarm} - 1, Nwin - 2) \tag{7.27}$$

This means that, considering the last alarm was i time steps ago, if the summation of the run-length probability up to $i - 1$ is higher than a threshold, it is considered that a ChangePoint was suffered in the last $i - 1$ time steps. If last alarm was more than $Nwin - 1$ time steps ago, then the summation in the run-length distribution goes up to $Nwin - 2$.

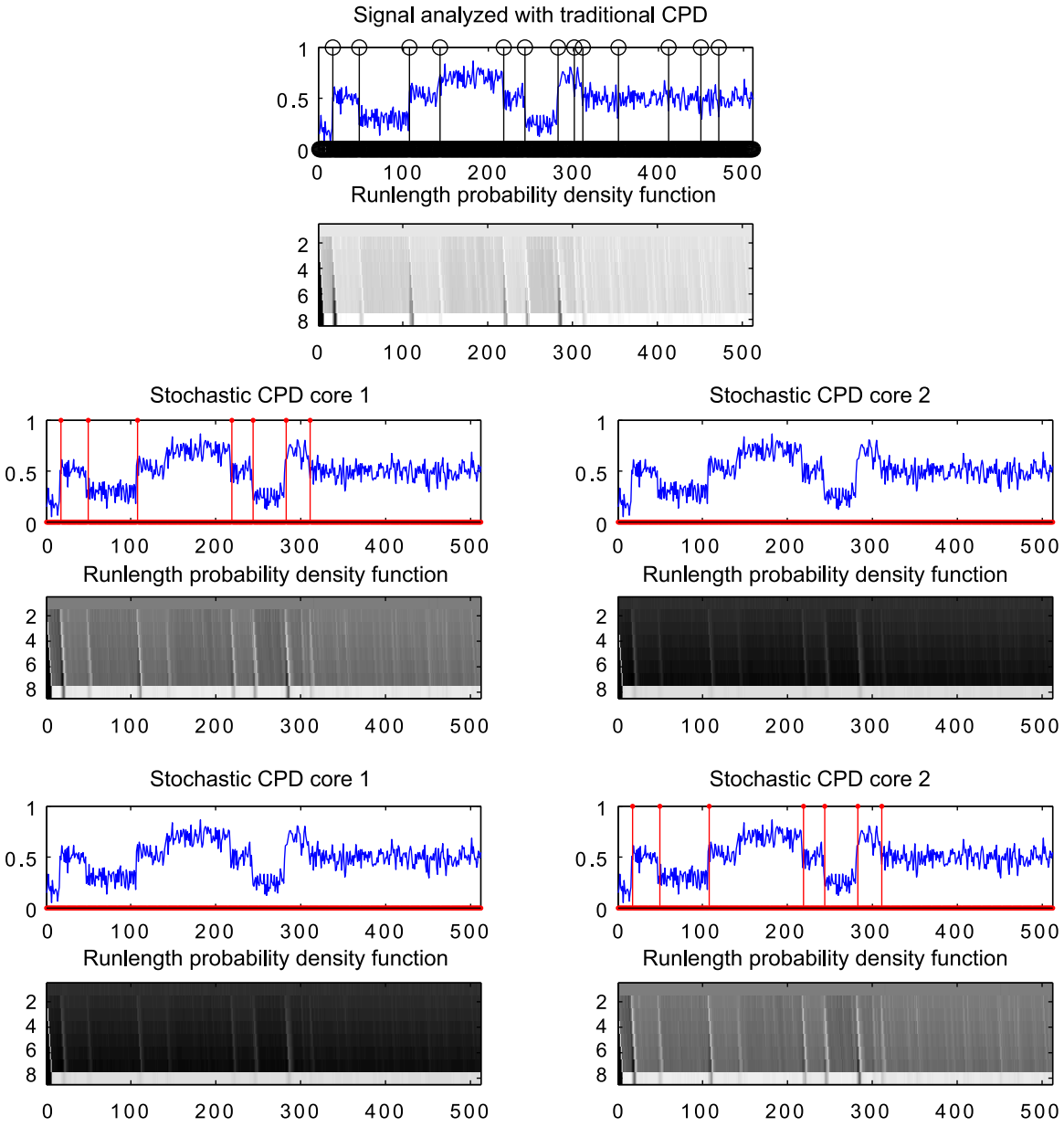


Figure 7.8: GF1 & GF2 chip test results. Same input is sent to the four CPD cores. The input signals is analyzed with the traditional CPD algorithm using a time window of eight time steps ($Nwin = 8$) and a computational time of 2048 clock cycles. The outputs of the four cores are shown. In red, the points of change are remarked. The identification of the points of change is done online.

7.5 Stochastic CPD Test Chip GF3

7.5.1 Changes in GF3

For GF1 and GF2 chips, the area used for each *CPD* core, seen in Figure 7.4, was considerably larger than in the design being presented here, $468\mu m$ by $468\mu m$ compared to less than $200\mu m$ by $200\mu m$. The reason for this, is that the new GF3 chip is not as programmable as its previous two versions. The idea behind these first two chips was to explore different parameter values when processing real signals, so that a new *CPD* chip could be tailored accordingly. Two of the most important parameters that were decided to be fixed were, the maximum run-length r_{max}, fixed to 5 after analyzing up to a value of 10, and the computational time, for which 4096 clock cycles was found to be sufficient for each cycle processing a new incoming sample.

In deciding what run-length was the most suitable one, testing signals were generated and the *CPD* algorithm was applied to them. In doing so, the decision threshold values triggering alarms were spanned from 0.3 to 0.95, using 0.05 steps, testing run-lengths from 3 to 10. For each threshold and run-lengths value, a probability of hit and miss was found, as well as the number of false alarms with respect to the real number of transitions in the mean of the signals. In choosing a threshold, so that the number of false alarms was not more than 10% of the number of real mean transitions for input signal X_t, the numbers in Table 7.2 were obtained. For the false alarms, a number of 1.0 would mean that the number of false alarms is equal to the number

Maximum run-lengths (Nwin)	false alarms	P_{hit}	P_{miss}
3	0.100	0.4704	0.5296
4	0.100	0.6796	0.3204
5	0.1046	0.8173	0.1827
6	0.1028	0.9053	0.0947
7	0.0992	0.9454	0.0546
8	0.1032	0.9704	0.0296
9	0.0958	0.9775	0.0225
10	0.1063	0.9866	0.0134

Table 7.2: Choosing the maximum run-length for the stochastic CPD. Number of false alarms, probability of hit and probability of miss for when the maximum run-length is varied.

of real transitions in the mean of the signals. Keeping in mind that the maximum run-length is desired to be as small as possible because the chip area scales proportionally to it (see Figure 7.4), a 0.8173 probability of hit was considered to be enough for the case of $r_{max} = 5$, and considering that this algorithm will be used for image processing, spatial median filters would take care of the misses.

Additionally, a maximum computational time for the stochastic processing of each incoming sample X_t had to be chosen. Chips GF1 and GF2 were then used to test the accuracy of the stochastic *CPD* processor using different computational times. The results are presented in Table 7.3. For both 2048 and 4096, less than 10% of false alarms was found, and a higher than 0.75 probability of hit. These two processing times were the ones considered more attractive. Finally, 4096 was chosen, but this choice did not mean that less computational time cannot be used if desired, in fact GF3 chip has the capability of choosing among four different computational times,

Computational Time	false alarms	P_{hit}	P_{miss}
512	0.0676	0.7056	0.2944
1024	0.1028	0.7169	0.2831
2048	0.0944	0.7549	0.2451
4096	0.0972	0.7732	0.2268
8192	0.1366	0.7845	0.2155
16384	0.1338	0.8000	0.2000
32768	0.1437	0.8113	0.1887
65536	0.1408	0.7986	0.2014

Table 7.3: Choosing the maximum computational time for the stochastic CPD. Number of false alarms, probability of hit and probability of miss for when the maximum computational time is varied. Chips GF1 and GF2 were used in the generation of these values.

512, 1024, 2048 and 4096. All the registers used in the stochastic encoding such as $\vec{Ro}$, Wo and $m\vec{u}o$ from Figure 7.4, and decoding registers $\vec{Ri}$, Wi and $\vec{mui}$ will then be 12 bits wide.

In the first two *CPD* chips GF1 and GF2, the state variables for each of the analyzed signal streams needed to be read and written from the chip for each new given sample X_t. This made the throughput of the chip very low (392 processed samples per second at 12Mhz). The read/write operation through an USB 2.0 interface would take $\approx 96.65\%$ of the used time at $12Mhz$. On the other hand now, with this new chip version, it was considered a good idea to store all the internal states on-chip so that higher number of samples could be processed per time unit, and lower power consumption could be achieved by limiting the I/O activity. In storing all of the internal states ($\vec{Ro}$, Wo and $m\vec{u}o$ registers from Figure 7.4), the SRAM memory blocks discussed in Chapter 6 were used. The usage of only up to metal 3 in the internal routing

	400mV	500mV	600mV	700mV	1200mV
Min Period	$4063.126ns$	$559.403ns$	$123.531ns$	$43.686ns$	$9.26ns$
Max Frequency	$\approx 246kHz$	$\approx 1.8MHz$	$\approx 8.1MHz$	$\approx 22.9MHz$	$\approx 108MHz$

Table 7.4: Maximum operating frequencies for GF5. Different voltages supplies are used at 27C.

of these SRAMs, and the possibility of lowering the voltage supply down to 400mV, made these SRAM memories very tempting to incorporate in the design. This new chip was then synthesized for a voltage supply of $500mV$ running at 1.8Mhz. With a design already synthesized, from which a netlist has been extracted, the maximum frequency of operation can be calculated for different voltage supplies by changing the timing files of the standard cell library. Table 7.4 shows the maximum frequency of operation obtained for the CPD architecture in GF3 for different voltage supplies at 27C.

Two more features were added to this new chip. In GF1 and GF2, with each processed sample X_t, an updated run-length probability distribution would be calculated, and it is off-chip that this probability distribution is analyzed for making the decision if an alarm should trigger or not. For GF3, access to the run-length distribution was not an option, and the decision of a Change-Point is performed on-chip in a stochastic manner. Additionally, a pre-normalizing unit was incorporated to the design, giving the option of pre-conditioning the signal that goes into the *CPD* cores.

In GF1 and GF2, $1mm^2$ of area was available in $65nm$ and $55nm$ GF process respectively. For GF3, $55nm$ process was used, with a chip size of $3.5mm$ by $3.5mm$,

increasing the area to $12.25mm^2$. With so much more area available, flat *Place & Route* was not an option any more. Chips GF1 and GF2 would take 4 hours for their *Place & Route*, and now with $12.25mm^2$ available, 12.25 x 4 hours was considered too much time for designs that had to be iterated several times. Consequently, hierarchical synthesis, also known as chip partitioning, had to be performed for this chip for the first time, where a unit in the design (in this case *CPD*_cluster block) was built separately, and then many copies of it were placed and interconnected in the final design.

For this chip, power consumption was a concern, and so it was decided to share the random number generators with as many *CPD* processing units as possible. The number of random number generators needed for each of the *CPD* cores is 17, and with a computational time of 4096 clock cycles, $17.log_2(4096) = 204$ signals needed to be routed to each *CPD* core. A reduction to the number of random number generators was attempted by analyzing correlation between signals in the architecture. A more in-depth explanation of the architecture will be addressed later, but at this point it is worth mentioning that there are 48 *CPD* cores in the design, so $48x204 = 9792$ wires would have to be routed all over the chip if the same random numbers were decided to be used for all of the cores. This is one of the reasons the decision of partitioning the design into clusters was made, where each of the clusters contains four *CPD* cores, and they locally share the same random number generators. Another reason for not sharing the same random numbers among all of the *CPD* cores, was that all of the

cores would not be necessarily processing all at the same time. Consequently, having all of those random numbers shared would mean that the LFSRs would have to be running all the time even if only one *CPD* core is used. This would make these high fanout lines distributing the LFSRs' constantly switch, increasing power dissipation.

A previous run with testing structures for the ultra-low voltage *SRAM* was not available (GF5 from Chapter 6 was actually fabricated after GF1, GF2 and GF3), and so it was decided to sacrifice a very small portion of the chip area to do this. Not all the memory blocks were placed to be tested in this area, only the *SRAM* blocks with a word size equal to 8 bits. The *SRAM* memory sizes used were *64x8*, *128x8*, *256x8* and *512x8*. The main reason for exploring different row sizes and not different word sizes was that the change in the word size does not impact in the memory speed as much as the number of rows does. The SRAM memory blocks used in this chip were the first iteration, GF5 was actually the second iteration, where some of the internal drivers were resized for a better speed performance. In testing these memory blocks access to all of the ports of the four placed *SRAMs* was given, and an additional signal that permits the selection of one of the four memory blocks. In Table 7.5 the ports for testing the memory blocks are listed. Figure 7.9 presents the whole GF3 chip layout. The amount of area used for the *SRAM* testing represents only 2.35% of the synthesized area (not considering pads).

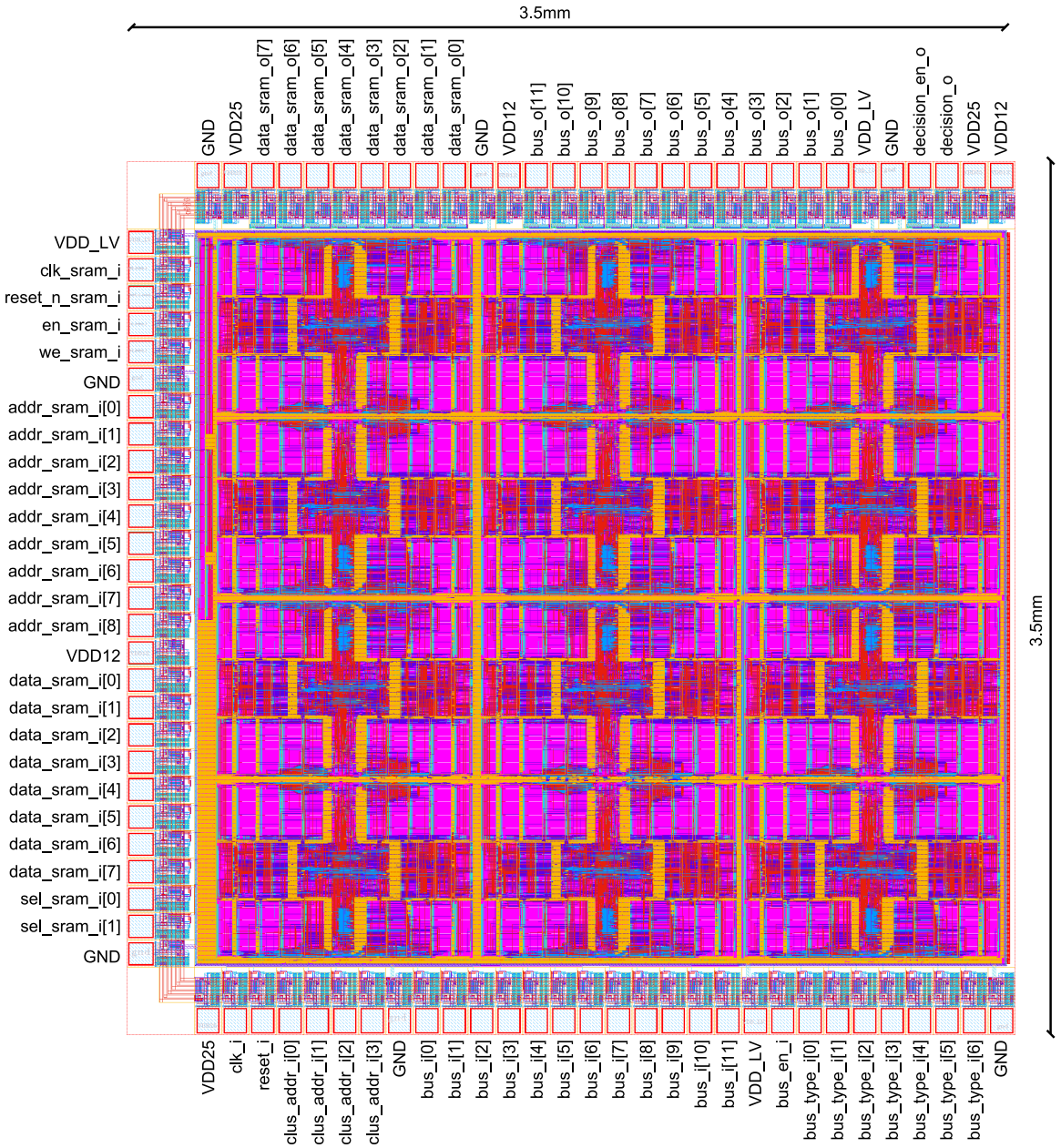


Figure 7.9: GF3 chip layout with pinout. This chip was fabricated in $55nm$ GF with $3.5mm$x$3.5mm$ in size. Pins connected to ground are named *GND*, pins connected to $2.5V$ are named *VDD25*, pins connected to $1.2V$ are named *VDD12* and pins connected to a voltage lower than $1.2V$ are named *VDD_LV*.

Signal name	Bits	O/I	Description
clk_sram_i	1	I	Clock input for the *SRAM* blocks.
sel_sram_i	2	I	This two-bit signal selects one of the four *SRAM* blocks.
reset_n_sram_i	1	I	Synchronous reset for the asynchronous controller inside of each of the *SRAM* blocks.
en_sram_i	1	I	Chip enable signal. If it is low, then no read or write operations can be performed. When it is high and we_sram_i is low, a read operation is performed.
we_sram_i	1	I	*SRAM* write enable. In order to write en_sram_i needs to be also high.
addr_sram_i	9	I	Input address for the *SRAM* blocks. The 512-row memory block will use all the bits, but the other blocks will just use the LSBs.
data_sram_i	8	I	Data input for the memory blocks. This signals goes to all of the *SRAM* blocks.
data_sram_o	8	O	Data output.

Table 7.5: Signals used for testing the *SRAM* blocks is GF3.

7.5.2 Architecture Description

Twelve *CPD* clusters were placed in the design as seen in Figure 7.10, where the input and output signals communicating to the chip are also shown. Each of the clusters in the system has an address to which they answer when data is sent to the chip. This address value is set by the cluster block's hardwired input *clus_cur_addr_i*. All the parameters stored on chip, as well as addresses and sample values, are sent through the input signal *bus_i*. The signal *clus_addr_i* determines the destination cluster for the data in *bus_i*, *bus_type_i* determines its purpose and *bus_en_i* enables the reception of data in the bus. All the programmable parameters in the system are local to the clusters. In Table 7.6 a brief explanation of all the interface signals for this block is shown. Only two single bit signals are received from the chip, *decision_o*

Signal name	Bits	O/I	Description
clk_i	1	I	Clock signal.
reset_i	1	I	Reset for the asynchronous controller in the *SRAM*.
clus_addr_i	4	I	Destination cluster for the data in *bus_i*.
bus_i	25	I	Input bus through which samples, addresses and parameter values are sent to each cluster.
bus_en_i	1	I	Enable for signal *bus_i*.
bus_type_i	6	I	Purpose for the data in *bus_i*.
decision_en_o	1	O	Flag that indicates when data in *decision_o* is valid.
decision_o	1	O	Change-Point decision output.

Table 7.6: Description of the CPD block *NORM_CPD_top* interface signals.

that indicates the Change-Point decision value, and *decision_en_o* that determines when the decision value is valid.

Due to the fact that samples are sent to the chip sequentially, and by knowing the amount of time spent processing by the chip, results will be received in a first-in-first-out basis. This makes redundant the need of an identifier for the events coming out of the chip. This processing methodology allows to *OR* together all of the *decision_o* and *decision_en_o* signals from each of the clusters in Figure 7.10.

Figure 7.11 shows that each cluster is composed of four *NORM_CPD_unit* blocks, where each of those units has a single *NORM* and *CPD* processing unit. The first one corresponds to the unit that normalizes the input signal (mainly using a subsampled second order IIR filter), which can be bypassed, and the second one is the *CPD* architecture unit based on Figure 7.4. Detailed explanation of input and output signals is not provided because there is practically no difference between this block's

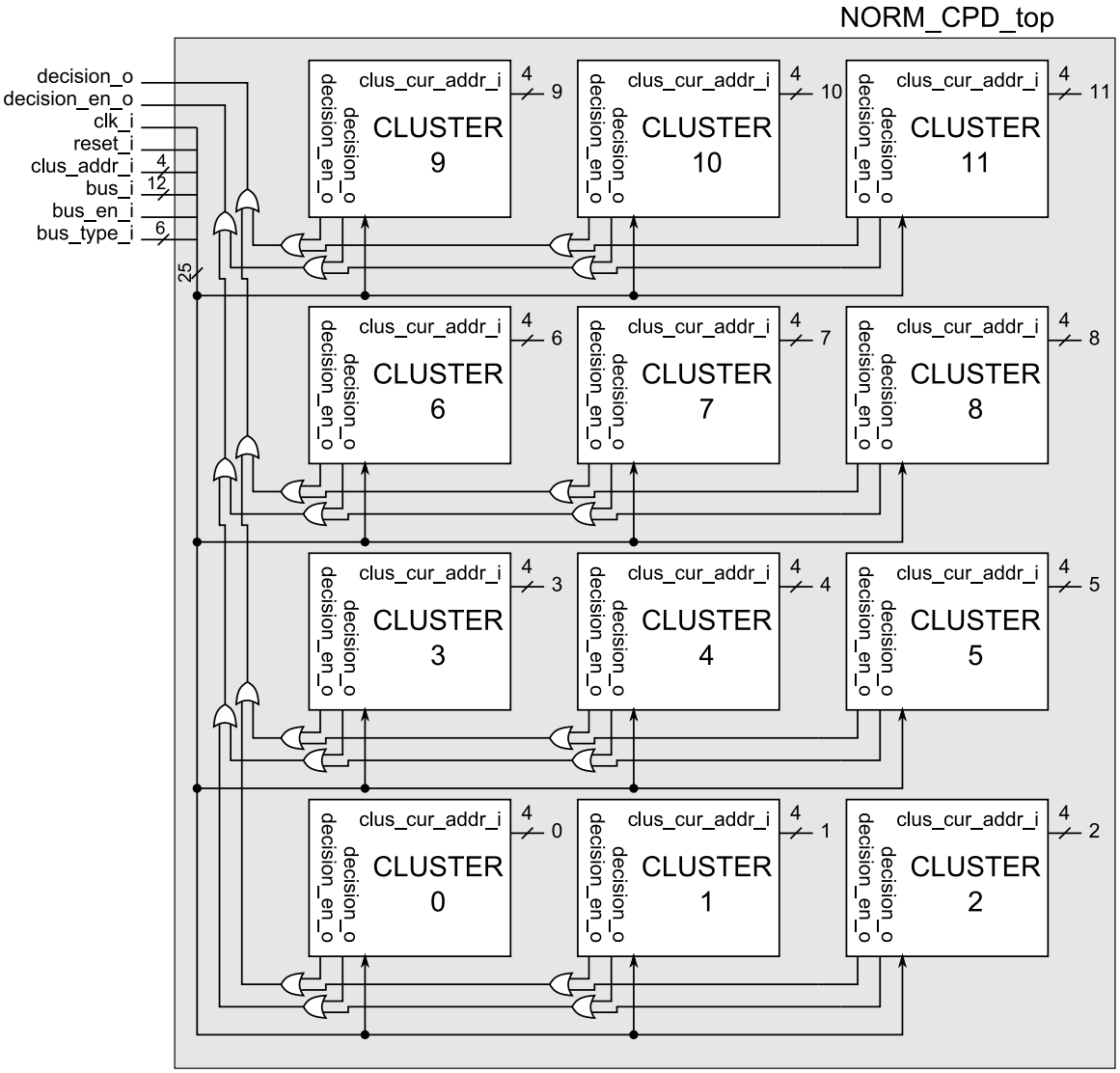


Figure 7.10: CPD cluster division for GF3. CPD block *NORM_CPD_top* was composed of 12 clusters of 4 CPD cores each. The cluster division can be visually mapped to the layout seen in Figure 7.9.

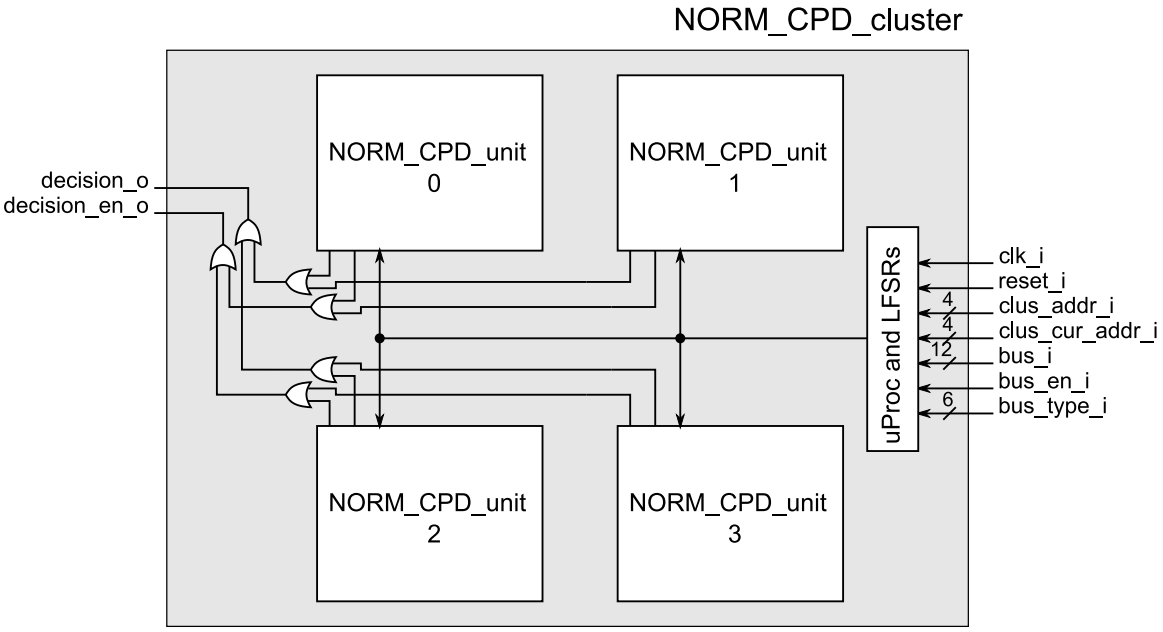


Figure 7.11: Block *NORM_CPD_cluster* is composed of four different *NORM_CPD_unit* blocks.

interface and the *NORM_CPD_top* block. It is at this level that the programmable parameters and random number generators are shared among the four units. As it will be seen later, each of the *NORM_CPD_unit* blocks will contain enough SRAM to keep in memory the states of up to 64 independent signals each, allowing a total of $64x12x4 = 3072$ independent analyzed signals that can now be used to process a small patch in a sequence of images. In Table 7.7 each address *bus_type_i* with the expected variable is shown.

The different areas for the blocks in the GF3 chip are presented in Table 7.8. Figure 7.12 presents a more visual comparison of the areas involved. The comparison is done between a single cluster in GF3 to the four CPD cores present in both GF1 and GF2 chips. The previous chips did not have a normalizing unit for each of its

Variable	Bits	Address	Description
LFSR_seeds	12	0 to 16	Seeds for the 17 LFSRs present in each of the clusters.
int_time	2	17	Register that configures the computational time among 512, 1024, 2048 and 4096 clock cycles.
mu0_val	12	18	This is the mean μ_0 for the Gaussian prior used for the mean of the input signal streams in the *CPD* algorithm.
w_val	12	19 to 22	Weights used for updating the mean variable states in the *CPD* algorithm.
b_val	12	23 to 27	Bernstein coefficients used for approximating the Gaussian distribution.
NburstDown	3	28 to 32	Scaling value applied to the input of the Bernstein polynomials approximator for the Gaussian distribution.
Nburst	4	33 to 37	Scaling value applied after the NburstDown scaling to the input of the Bernstein polynomials approximator for the Gaussian distribution.
h_val	12	38	Parameter in the *CPD* algorithm (see Equation 7.2)
t_val	12	39	Threshold used in the decision-making process for finding Change-Points.
accum	3	40	*NORM* processing unit parameter.
filterCoeff	10	41 to 46	*NORM* processing unit parameter.
Gain	9	47	*NORM* processing unit parameter.
norm_mean	1	48	*NORM* processing unit parameter.
use_norm	1	49	Value that determines the usage of the pre-conditioning of the input signals.
signal_address	10	50	Address that targets one of the $64x4$ signals that the four *NORM_CPD_unit* block can process.
sample	1	51	Sample from one of the 64 independent streams a *NORM_CPD_unit* block processes.

Table 7.7: Expected values at the input bus *bus_i* for each address in *bus_type_i*.

Block	Area
NORM_CPD_cluster	575090 μm^2
CPD_unit LOGIC	29537.32 μm^2
CPD_unit SRAM	68899.68 μm^2
NORM_unit LOGIC	4553.62 μm^2
NORM_unit SRAM	40781.88 μm^2
CPD processor	810483 μm^2 (GF1 & GF2)

Table 7.8: Comparison of block areas in GF3. The last value in the table corresponds to the area of the GF1 and GF2 chips without considering the pads' area.

CPD cores, so a fair comparison would be between a *CPD processor* from GF1 or GF2, and the *CPD_unit LOGIC* from GF3, which shows a reduction of 6.85 times in area. In the decrease from 16 to 5 in the maximum run-lengths from chips GF1 and GF2 to GF3, a reduction to $100x5/16\% = 31.25\%$ from the previous chips is accomplished. If additionally it is considered that a maximum computational time in GF1 and GF2 of 2^{24} was reduced to 2^{12}, an additional cut in the area used by half is achieved when going from 24 bit registers to 12 bit registers. The reduction estimate in area for each CPD core for this new chip is $31.25/2 = 15.625\%$, which corresponds to a reduction of 6.4 times, which is pretty close to the previously obtained empirical values.

As seen in Figure 7.13, each of these units has its corresponding CPD core communicating to a local *SRAM*. All of the *SRAMs* in this chip will have 64 rows, meaning that each of the *NORM* and *CPD* cores is multi-tasked among 64 independent signal streams. This makes it possible to process and keep all the internal states of

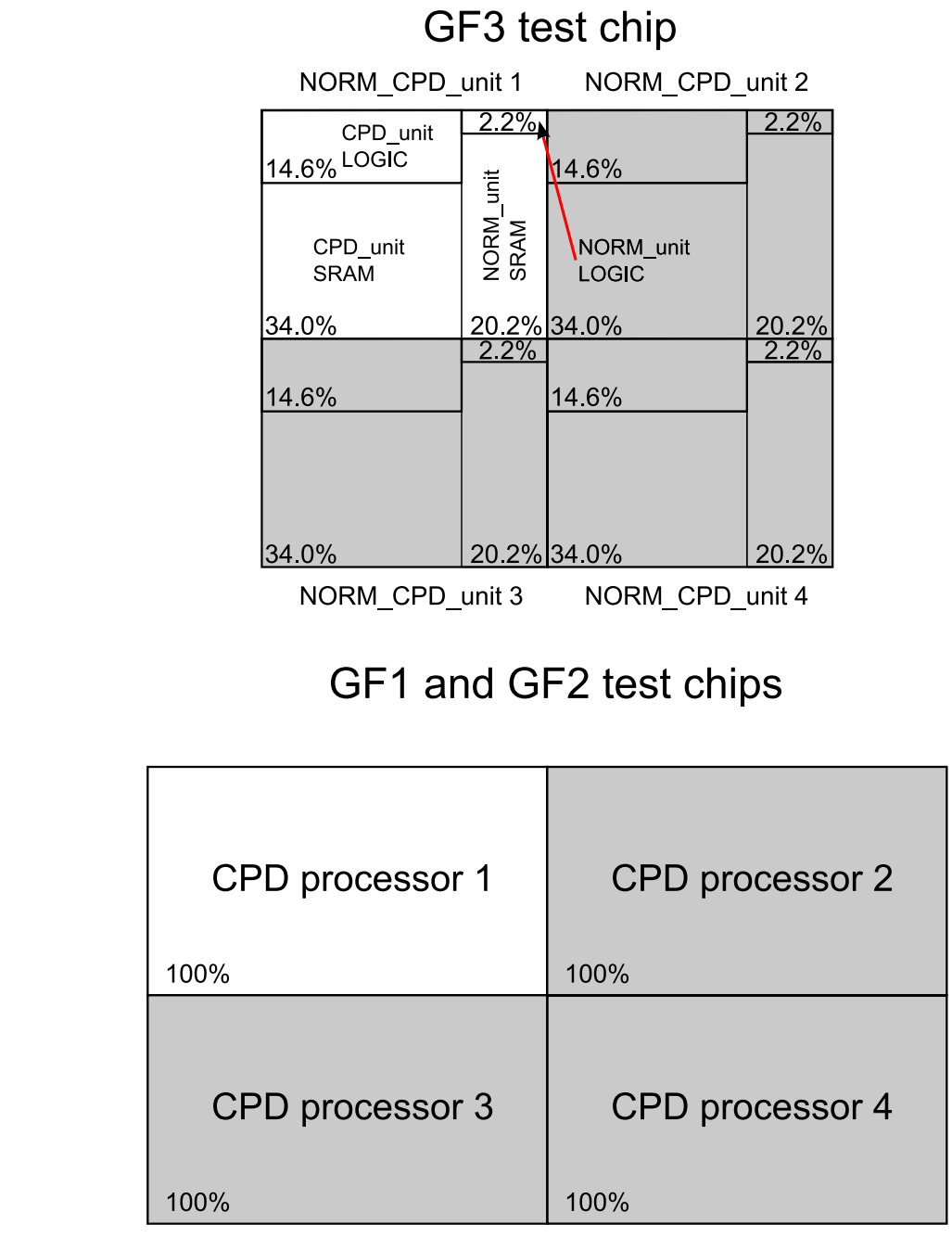


Figure 7.12: Area comparison for chips GF1 & GF2 vs GF3. On the left GF3, and on the right GF1 & GF2. A cluster containing four of the CPD cores in GF3 is compared to GF1 & GF2 chips. The areas presented do not consider the area from the pads.

$64x4x12 = 3072$ signal streams, or one can think of it as a $64x48$ pixel images. Considering the chip running at 1.8Mhz for $500mV$, approximately $\approx 7fps$ can be accomplished for 64 x 48 pixel images. Keeping in mind that the amount of data to be sent and retrieved from the chip is dramatically less than in the previous versions (only the single bit Change-Point decisions are streamed out of the chip), one can estimate that this design will process approximately 360 times faster than the previous versions at the same clock frequency. In the top left corner of Figure 7.13, four signals are responsible of the processing of a sample X_t. For each sample transaction, first an address through input *bus_i* will point to one of the 64 independent streams to which the incoming sample corresponds, along with a write enable signal *addr_we_i*. Input *use_norm_i* will determine if the processed signals are pre-conditioned or not, routing the sample to either the *CPD* or *NORM* unit. If the sample if routed to the normalizing block, after the normalizing process takes place, the output of this block (signal y) is sent through a mux to the *CPD* unit. A new sample is accepted through *bus_i* when signal *x_we_i* is asserted.

A block diagram of the *CPD_unit* is presented in Figure 7.14. Two main blocks can be distinguished, the memory block, where all the state variables are stored, and the *CPD_core* block. The shown *LOGIC* coordinates the communication between these two blocks. The memory block is built up out of four $64x32$ *SRAMs* simulating a $64x128$ *SRAM*.

The previous CPD chip versions had the capability of programming the amount

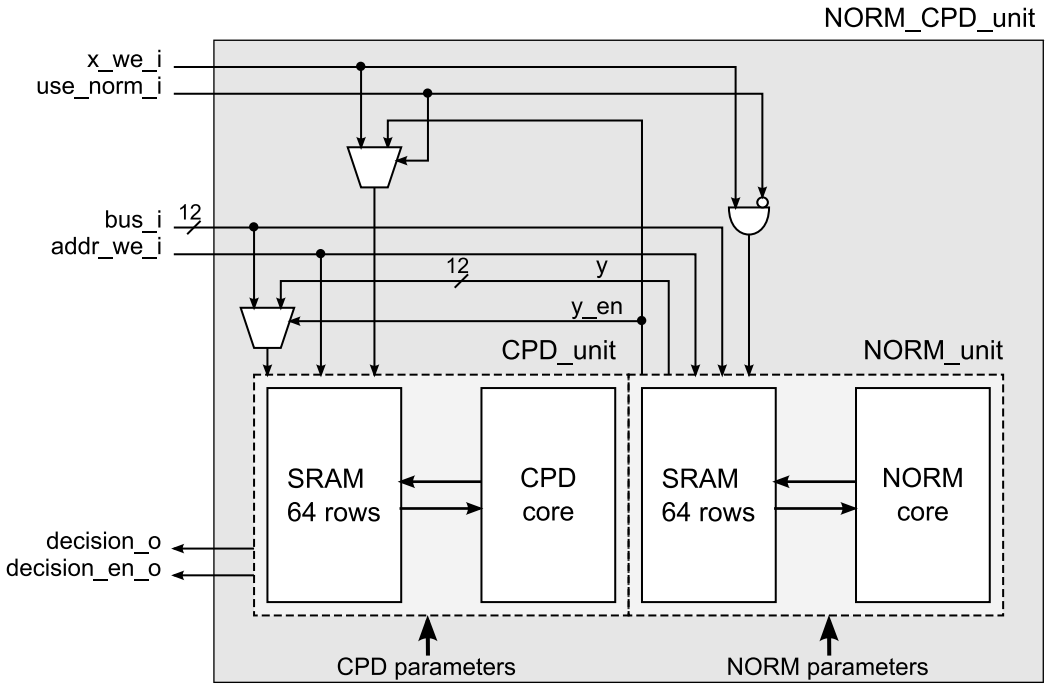


Figure 7.13: Internal division of the *NORM_CPD_unit* block.

of time used for processing each new sample, and for each choice, all LFSRs were reprogrammed accordingly so that their period was equal to the chosen processing time. For this chip, LFSRs were fixed to 12 bits, with a period of 4095 clock cycles. Changing the computational time is still a capability of this chip, but in a more constrained way. The input signal *int_time_i* is a two-bit signal that can configure the processing time to be 4096, 2048, 1024 or 512 clock cycles, but the choice of this processing time does not change the fixed-size 12 bits LFSRs. This signal programs an internal counter that generates a processing enable signal.

It is through the input bus signal *bus_i* that the address and the sample value for one of the 64 independent signals is sent. For both the address and sample value, two enable signals are provided *x_we_i* and *addr_we_i*. An address is first received through

the bus *bus_i* with an enable pulse through *addr_we_i*. This operation reads from the *SRAMs* the state variables for the address provided, and makes these values available for the *CPD_core* block. After this, when a sample is received through the same bus accompanied by a pulse through the *x_we_i* input, an internal counter resets and the block begins to process for the amount of time set by the input signal *int_time_i*. When processing finishes, a decision is taken on weather a change in the mean value of the signal happened or not (a Change-Point). The decision is signaled with a pulse through *decision_en_o*, and the decision value in *decision_o*. It is through a feedback of the pulse in *decision_en_o* that the new state variables are written in memory. The state variables held in memory are the run-lengths probability distribution *Ro* (five 12 bits values) with the normalizing constant *Wo* (12 bit value), the mean values *Muo* (four 12 bits values) and the variable *last_alarm* that is used in the decision-making of a Change-Point. In Table 7.9 a brief description of the input and output signals is provided.

An updated CPD architecture is shown in Figure 7.15 considering a maximum computational time of 4096, a maximum run-lengths of 5, and a maximum number of Bernstein coefficients of 5 as well. It can be observed that on the top right corner a few changes were made to the architecture. That change is due to the on-chip decision calculation for Change-Points. When a Change-Point is detected, a counter called *last_alarm* is set to zero, and with every new arriving sample X_t, that counter is increased until a new Change-Point is reached. When this counter reaches the

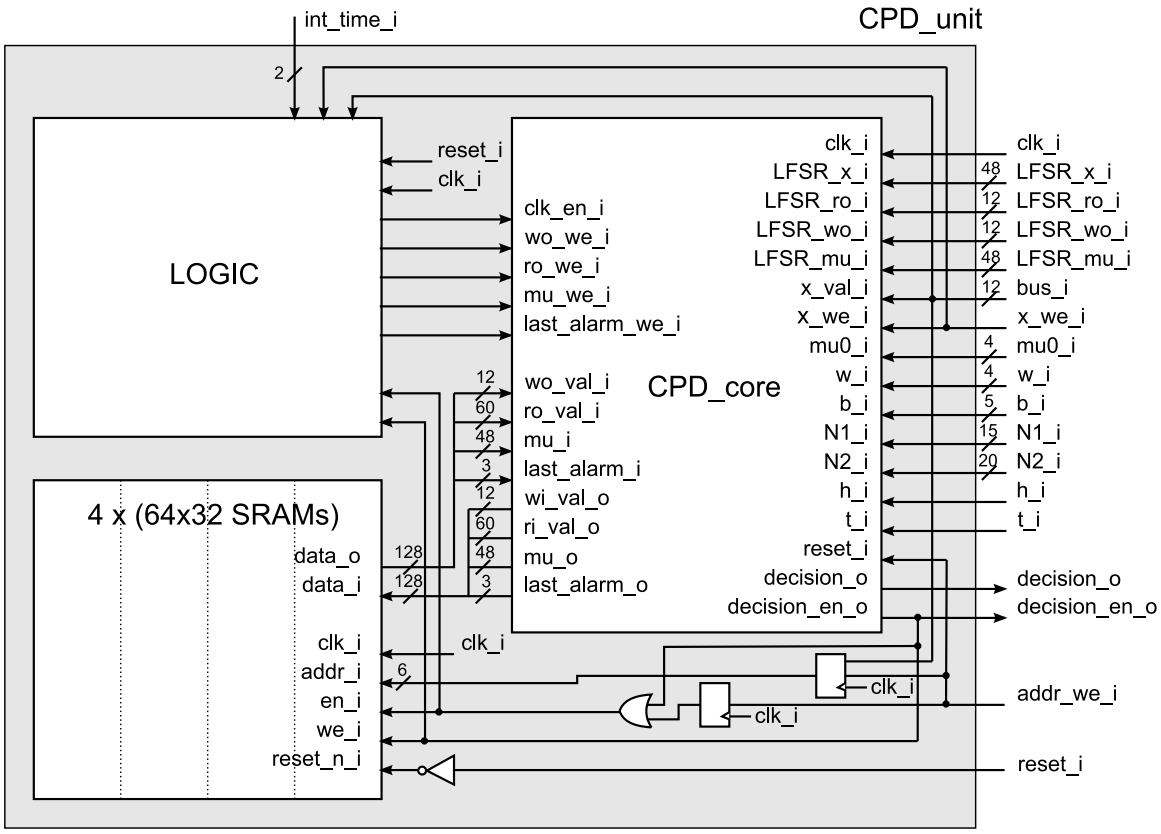


Figure 7.14: *CPD_unit* block diagram.

value corresponding to the maximum run-length minus 2 (in this case 3), with each new sample that doesn't trigger a Change-Point decision, the counter value remains unchanged. The run-length probability distribution $P(r_t|X_{1:t})$ for the case of the chosen maximum run-length of 5, can be evaluated with the integer values 0 to 4. The probability value

$$\sum_{i=0}^{last\_alarm} P(r_t = i|X_{1:t}) \tag{7.28}$$

corresponds to the probability that the Change-Point happened between the previous positive decision of Change-Point and the present time. It is that probability the one

Signal name	Bits	O/I	Description
clk_i	1	I	Clock inout.
reset_i	1	I	Reset for the *SRAM* asynchronous controller.
LFSR_x_i	12	I	Random numbers used for encoding samples.
LFSR_ro_i	12	I	Random numbers used for encoding the run-lengths probability values.
LFSR_wo_i	12	I	Random numbers used for encoding the run-lengths probability normalizing constant.
LFSR_mu_i	12	I	Random numbers used for encoding the mean values.
bus_i	12	I	Bus for sending addresses and samples into the unit.
x_we_i	1	I	Write enable for an input sample.
addr_we_i	1	I	Write enable for the input address.
int_time_i	2	I	Signal that determines the processing time spent for each new sample. For *int_time_i = 0* the processing time is 512 clock cycles, for *int_time_i = 1* 1024 clock cycles, for *int_time_i = 2* 2048 clock cycles and for *int_time_i = 3* 4096 clock cycles.
mu0_i	1	I	This is the mean μ_0 for the Gaussian prior used for the mean of the input signal streams in the *CPD* algorithm. This is a signal that has already been encoded sthocastically.
w_i	4	I	Weights used for updating the mean variable states in the *CPD* algorithm. These signals have been already encoded stochastically.
b_i	1	I	Stochastically encoded Bernstein coefficients used for calculating the Gaussian distribution.
N1_i	3	I	Burst down value for the Bernstein blocks in *CPD_core*.
N2_i	3	I	Burst value for the Bernstein blocks in *CPD_core*.
h_i	1	I	Stochastically encoded value for h in the *CPD* algorithm (see Equation 7.2).
t_i	1	I	Stochastically encoded threshold for the decision-making algorithm applied on the run-lengths distribution.
decision_o	1	O	Change-Point decision value.
decision_en_o	1	O	Enable for the Change-Point decision value.

Table 7.9: Description of the *CPD_unit* signals.

used in the comparison with a threshold to make the decision of a Change-Point. Since there is a maximum run-length, meaning that the distribution from the original *CPD* algorithm has been truncated, the probability $P(r_t = r_{max} - 1|X_{1:t})$ represents the

probability of Change-Point for the run-lengths $r_{max} - 1$ and all higher run-lengths. This means

$$\sum_{i=r_{max}-1}^{+\infty} P'(r_t = i|X_{1:t}) = P(r_t = r_{max} - 1|X_{1:t}) \tag{7.29}$$

where P' is the non-truncated *CPD* run-length distribution. If track was not kept for the previous Change-Points with the register *last_alarm*, a trigger decision of a Change-Point could be set for several consecutive samples, even if there was only one real change in the mean of the analyzed signal.

Finding the probability of Change-Point would consist of adding up the corresponding values from the $\vec{R}i$ registers after the incoming sample was processed based on the *last_alarm* value. This is done by masking the stochastic streams going to block *probability of change* in Figure 7.15. After this, a division by the value found in register Wi needs to take place. It was shown before how the division performed by JK flip flops, which is a low precision one, worked well in the normalization done in Figure 7.15. These flip flops not only help in normalizing, but they also prevent $\vec{R}i$ and Wi registers from slowly converging to all zeros as the run-length probability distribution is updated every time a new sample arrives. These JK flip flops were analyzed, but they were found not good enough to perform the accurate division required for the decision-making process. Architectures using Bernstein polynomial approximations for exponential and logarithmic functions were then explored, where a good accuracy was acquired using the logarithmic properties of the division. This architecture would work perfectly fine, but there was a substantial amount of logic

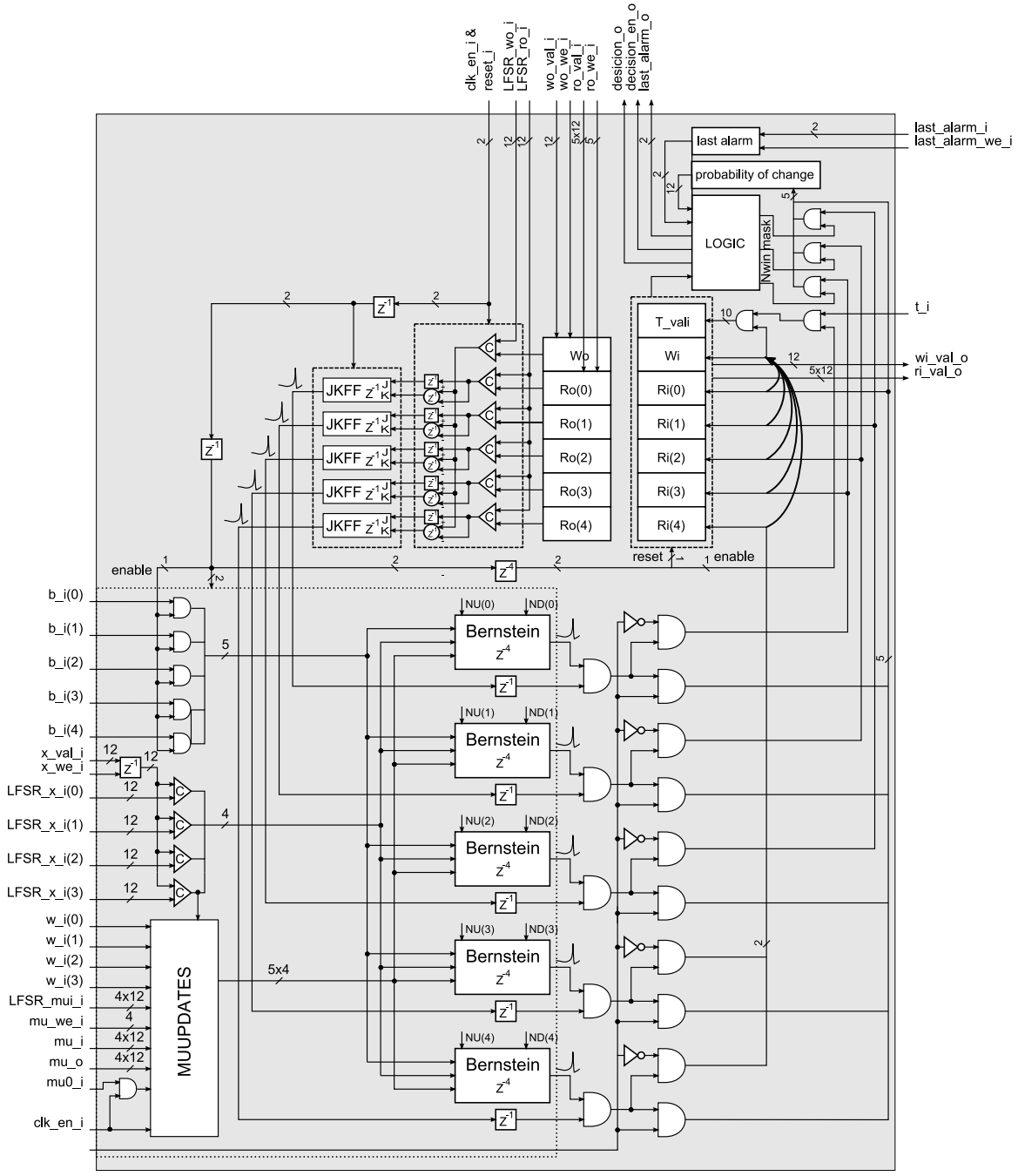


Figure 7.15: Updated architecture for the CPD processing unit in GF3. At the top right corner the part of the circuit calculating the generation of alarms (Change-Points).

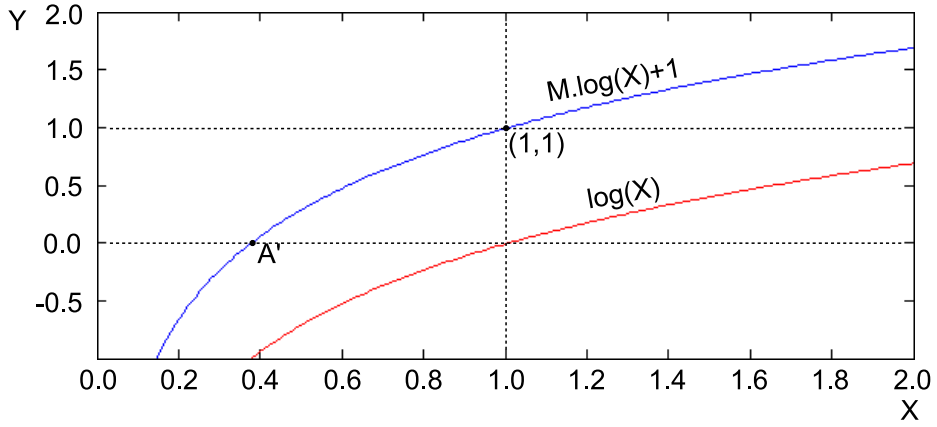


Figure 7.16: Plot of functions $log(x)$ and $Mlog(x)+1$.

added to the architecture.

The division x/y is desired to be performed, where $x \in [x_i; 1]$ and $y \in [y_i; 1]$. One can do $log(x/y) = log(x) - log(y) = (log(x) + 1) - (log(y) + 1)$. Let's observe curve $M.log(X) + 1$ from Figure 7.16, where $X \in [A'; 1]$ $Y \in [0; 1]$. Depending on the range desired for x and y, one can move point A' by introducing a scaling factor to the logarithmic function so that a Bernstein approximation can be used on it on the area defined by $(0, 0)$ and $(1, 1)$. In this design, for x, A' would be x_i, and for y, A' would be y_i. Let's find the coefficients M for x and for y:

$$M_x log(x_i) + 1 = 0 \Rightarrow M_x = -\frac{1}{log(x_i)}$$

$$M_y log(y_i) + 1 = 0 \Rightarrow M_y = -\frac{1}{log(y_i)}$$

One can now do the following:

$$\begin{aligned} log(x/y) &= \left(\frac{log(x)}{log(x_i^{-1})} + 1\right) log(x_i^{-1}) + 1 - log(x_i^{-1}) \\ &- \left(\left(\frac{log(y)}{log(y_i^{-1})} + 1\right) log(y_i^{-1}) + 1 - log(y_i^{-1})\right) \\ &= log(x_i^{-1})\left(\frac{log(x)}{log(x_i^{-1})} + 1\right) - log(y_i^{-1})\left(\frac{log(y)}{log(y_i^{-1})} + 1\right) + log(x_i/y_i) \end{aligned}$$

If now $x_i = y_i = a$ then:

$$log(x/y) = log(a^{-1})\left(\frac{log(x)}{log(a^{-1})} + 1 - \left(\frac{log(y)}{log(a^{-1})} + 1\right)\right)$$

If instead of log, one could use $log_{a^{-1}}$, then $log(x/y) = log_{a^{-1}}(x) + 1 - (log_{a^{-1}}(y) + 1)$, and both $p1 = log_{a^{-1}}(x) + 1$ and $p2 = log_{a^{-1}}(y) + 1$ can be approximated using Bernstein polynomials. Assuming that in the x/y calculation $x < y$, Figure 7.17 presents the block diagram for a stochastic divider. For this architecture an additional Bernstein approximation was done on the exponential function $(a^{-1})^{-w}$ for $w \in [0; 1]$. In this case $w = p2 - p1$, which belongs to $[0; 1]$.

Very accurate results were found utilizing the approach in Figure 7.17, but now many more random number generators would be required to perform the approximated calculation of the division. A much simpler solution was found, requiring little logic to be added to the original CPD stochastic architecture in Figure 7.4. This solution takes advantage of the stochastically encoded streams going into the $\vec{R}i$ and Wi

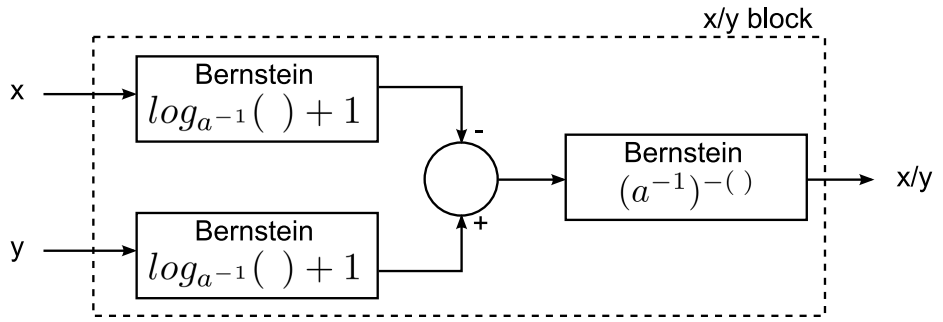


Figure 7.17: Division based on Bernstein approximations. Block diagram for a stochastic divider using Bernstein polynomial approximators on logarithms and exponential functions.

registers. The problem being addressed here is the division by the normalizing constant Wi, but what if instead of scaling the probability value $\sum_{i=0}^{last\_alarm} P(r_t = i|X_{1:t})$, one scaled the chosen threshold? This is the reason the input t_i was added to the block. This input provides an already stochastically encoded value for the programmed threshold. By just performing the AND operation between the incoming stream for Wi and the threshold stream, a scaled value for the threshold is decoded in the register T_vali. The register *prob_change* will accumulate the probability of Change-Point by masking the input streams for the registers $\vec{Ri}$ depending on the *last_alarm* value. After this, a simple comparison for values in registers T_vali and *prob_change* takes place for making the decision.

The number of coefficients used for the Bernstein approximations was fixed to five. In Figure 7.18 three different numbers of Bernstein coefficients were used to approximate half of a Gaussian bell with standard deviation 0.33. The search for the Bernstein coefficients was made taking into account that the Bernstein coefficients

Nber needed to be between zero and one, meaning that the best approximation could be found inside an *Nber*-dimensional hypercube where for each dimension only values $\in [0, 1]$ are considered. From Figure 7.18, the most area-efficient number of coefficients for the Gaussian approximation was found to be 5.

Figure 7.19 presents the block diagram for the *NORM_unit* block. At the core of the normalizing process are two filters, a downsampling filter and a second order *IIR* filter. Three *64x24 SRAM* blocks were used, and they were divided into two main blocks. The main reason for doing this division instead of using a single *SRAM* memory block, is for saving power. State variables stored in one of the *SRAM* blocks are updated with every new sample X_t, but the state variables of the other one are not.

Just like in *CPD_core* an address identifying the signal stream is first received in *bus_i* with an enable pulse at the input *addr_we_i*. This updates the output of the *SRAM* blocks that provide the state variables for the current sample to process. This address is also registered so that at the end of the processing, one still knows where to send the updated state variables. The state variables are the ones connected to the *data_i* and *data_o* ports in both of the *SRAM* blocks. At the end of the normalizing process of a sample, a pulse is sent through *y_en_o* and/or *state_en_o* and the new state variables are written in memory. For the case of the $2x(64x24SRAMs)$ there is a feedback from *data_o* to *data_i*, and that is because this *SRAM* block stores the sample values $x(n-1)$, $x(n-2)$, $y(n-1)$ and $y(n-2)$ from a second order *IIR* filter,

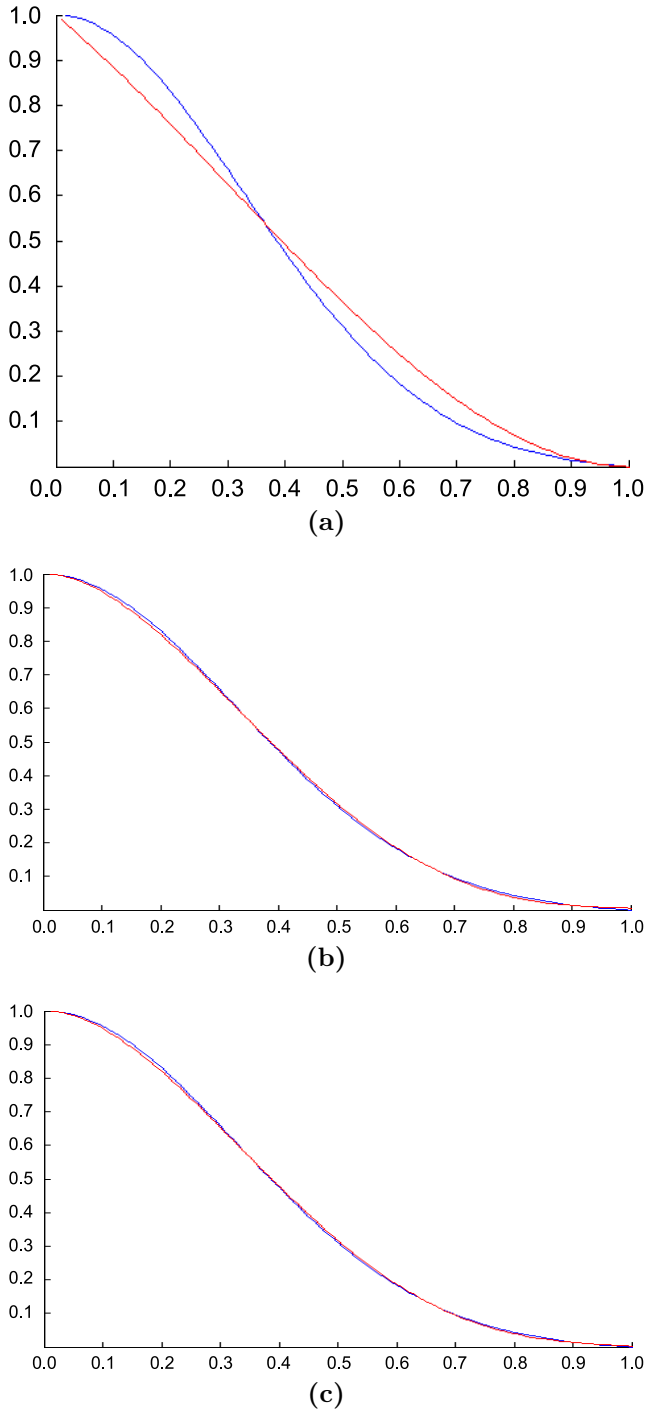


Figure 7.18: Approximation of the Gaussian bell using the Bernstein approach. Approximations for half a Gaussian bell of standard deviation 0.33 using different number of Bernstein coefficients. For 4, the area in between the two curves is 0.0022, for 5, $3.2411e^{-5}$ and for 6, $2.8694e^{-5}$.

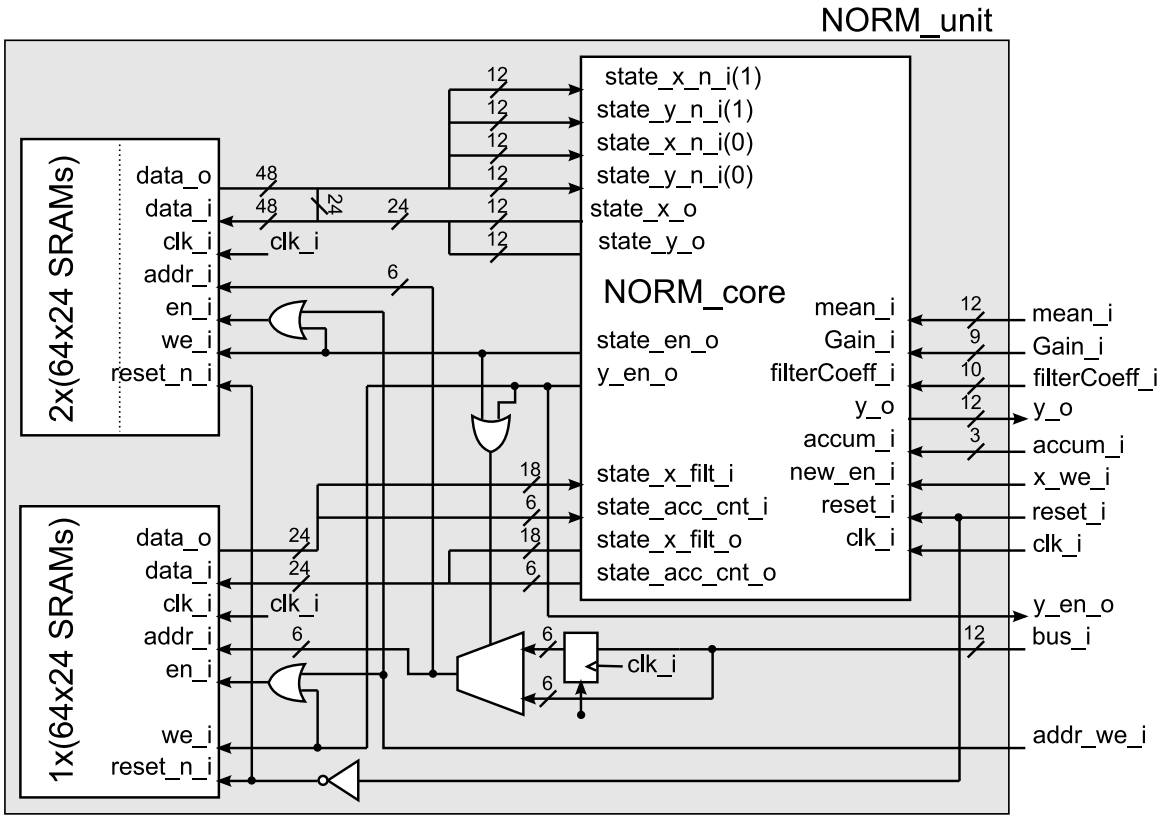


Figure 7.19: *NORM_unit* block diagram.

and when the new states are written, $x(n-1)$ is updated with *state_x_o*, $x(n-2)$ with $x(n-1)$, $y(n-1)$ with *y_state_o* and $y(n-2)$ with $y(n-1)$.

In this chip version capability of normalizing the input signals coming off-chip was introduced. This normalizing block, as mentioned before, can be bypassed if desired. The main reason for introducing this block is that sometimes the input signal suffers very small changes, and, in order to track those changes using the *CPD* algorithm, a normalization process needs to take place. An example of a normalization process is shown in Figure 7.20. The normalization block is conceptually explained in Figure 7.21. The computation of the normalizing block uses traditional computation

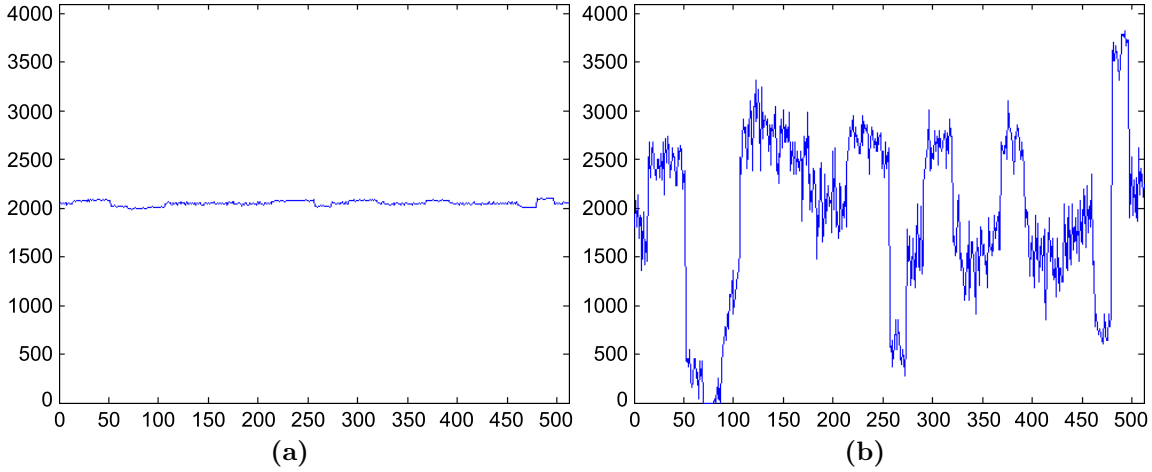


Figure 7.20: Example of the normalizing process in GF3. On the left an example of a signal without normalizing. On the right we see the signal on the left after the normalization process.

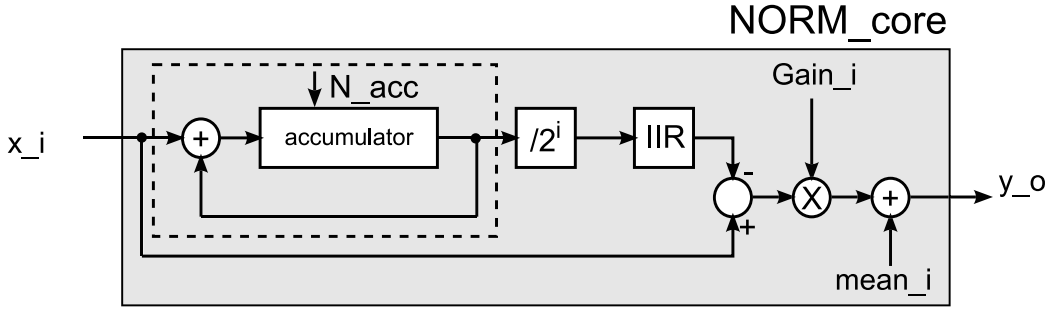


Figure 7.21: Conceptual block diagram for the *NORM_core* block.

structures, in comparison with the *CPD_unit* block.

When normalizing the input signal X_t, a long-term mean needs to be extracted. The original idea was to use *FIR* filters because of their very convenient characteristic of being linear in phase, but for the case at hand where a very slow converging filter is required, thousands of coefficients would be needed, making it non-viable. *IIR* filters were then considered, where even if one would suffer in the phase domain, the required filter characteristics would be achieved with just a few coefficients. A very

simple second order *IIR* filter was used in the architecture (general form for second order *IIR* filters in Equation 7.30), but it was noted that when talking about low-pass filters with such low converging rates, very high precision coefficients need to be stored. This would increase considerably the amount of logic allocated for this block, especially considering that a multiplier block had to be synthesized. A very high precision multiplier, even in the case of adding pipelining stages, would be the speed bottleneck of the whole system. A pre-filtering stage was then added to the design. In Figure 7.21, an accumulator would add *N_acc* number of consecutive samples, and a scaled down version of this value would be the one used as an input for the *IIR* filter. The combination of these two filters can provide the slow converging behavior desired. This first filter would just filter and subsample the input signal by *N_acc*, and this subsampled signal is then processed by the *IIR* filter. The output of the *IIR* filter would get resampled to the original frequency with a sample-and-hold, and this is the signal subtracted to the original signal *x_i*. After this, a gain stage is applied and the desired mean for the final signal output is added. This final addition of a desired mean was used because in the *CPD* algorithm one needs to supply the mean of the prior distribution, μ_o. The usage of a sample and hold to resample the filtered signal is not the best technique since it introduces frequency artifacts, but it is very simple and easy to implement, and it has proven to be effective. A brief description

Signal name	Bits	O/I	Description
clk_i	1	I	Clock input.
reset_i	1	I	Reset input.
mean_i	12	I	Programmable value. Mean for the output of the normalizing block *NORM_core*.
Gain_i	9	I	Programmable value. Gain applied to the difference of the input signal stream x_t and the low-pass filtered signal coming out of the *IIR* filter.
filterCoeff_i(0 to 5)	10	I	Programmable values. Coefficients a_0, a_1, a_2, b_0, b_1 and b_2 from Equation .
y_o	12	O	Normalized signal.
accum_i	3	I	Number of samples added together to generate the first low frequency downsampled signal in Figure 7.21.
x_we_i	1	I	Enable signal for the new arriving sample X_t.
bus_i	12	I	Bus for sending addresses and samples into the unit.
y_o	1	O	Enable signal for the y_o output.
addr_we_i	1	I	Write enable for the input address.

Table 7.10: Description of the *NORM_unit* signals.

of the inputs and outputs of the *NORM_unit* block is provided in table 7.10.

$$b_0 y(n) = a_0 x(n) + a_1 x(n-1) + a_2 x(n-2) - b_1 y(n-1) - b_2 y(n-2) \tag{7.30}$$

$$H = \frac{Y}{X} = \frac{a_0 + a_1 z^{-1} + a_2 z^{-2}}{b_0 + b_1 z^{-1} + b_2 z^{-2}} \tag{7.31}$$

Table 7.11 shows the maximum measured operating frequencies for the chip, along with the maximum frames per second obtained at different voltage supply values. A video corresponding to a traffic intersection was processed by the GF3 chip. The video is $720x640$ pixels in size, and it broken down to small $48x64$ pixel patches to be processed by the GF3 CPD chip. The video was taken from a drone hovering on top of that intersection, where vibrations in the set up would create the effect

	400mV	500mV	600mV	700mV	1200mV
Max Frequency	$200kHz$	$1.75MHz$	$7.9MHz$	$20.1MHz$	$95MHz$
Frames per second	$2.38fps$	$6.8fps$	30.7	$78.1fps$	$370fps$

Table 7.11: Measured clock speeds for GF3. Maximum clock operating frequency along with the number of frames per second for an image patch of 48x64 pixels for different voltage supplies.

of unwanted Change-Points from the algorithmic point of view. Even considering these vibrations, the processing of this video shows in Figures 7.22, 7.23 and 7.24 the successful recognition of cars moving in that intersection. Along an image processing pipeline, some filters, such as median ones, can be easily used to get rid of all of the salt and pepper noise.

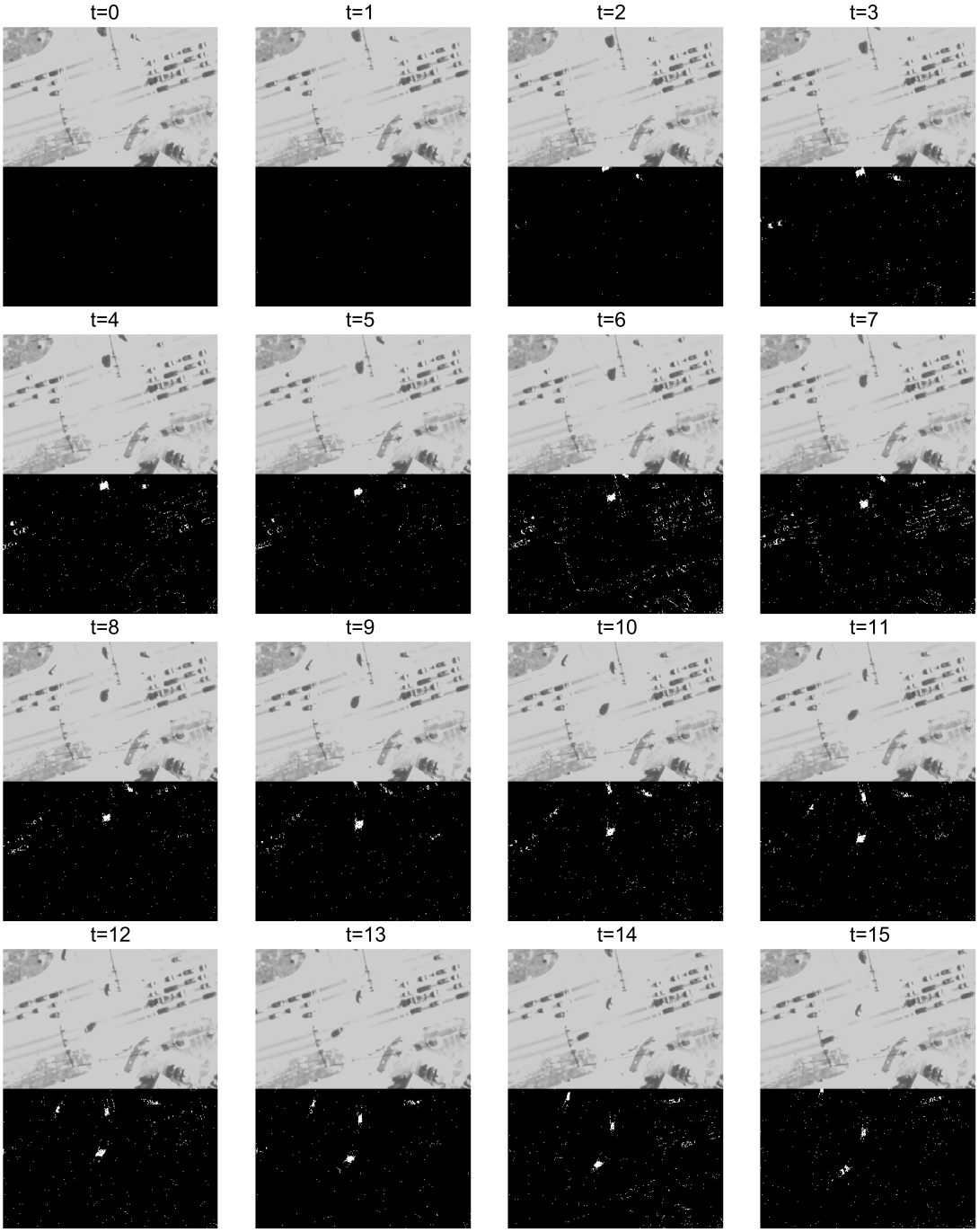


Figure 7.22: Video processed by the GF3 CPD chip (figure 1). For each time step, the top image is the greyscale image used as the input for the CPD processing algorithm.

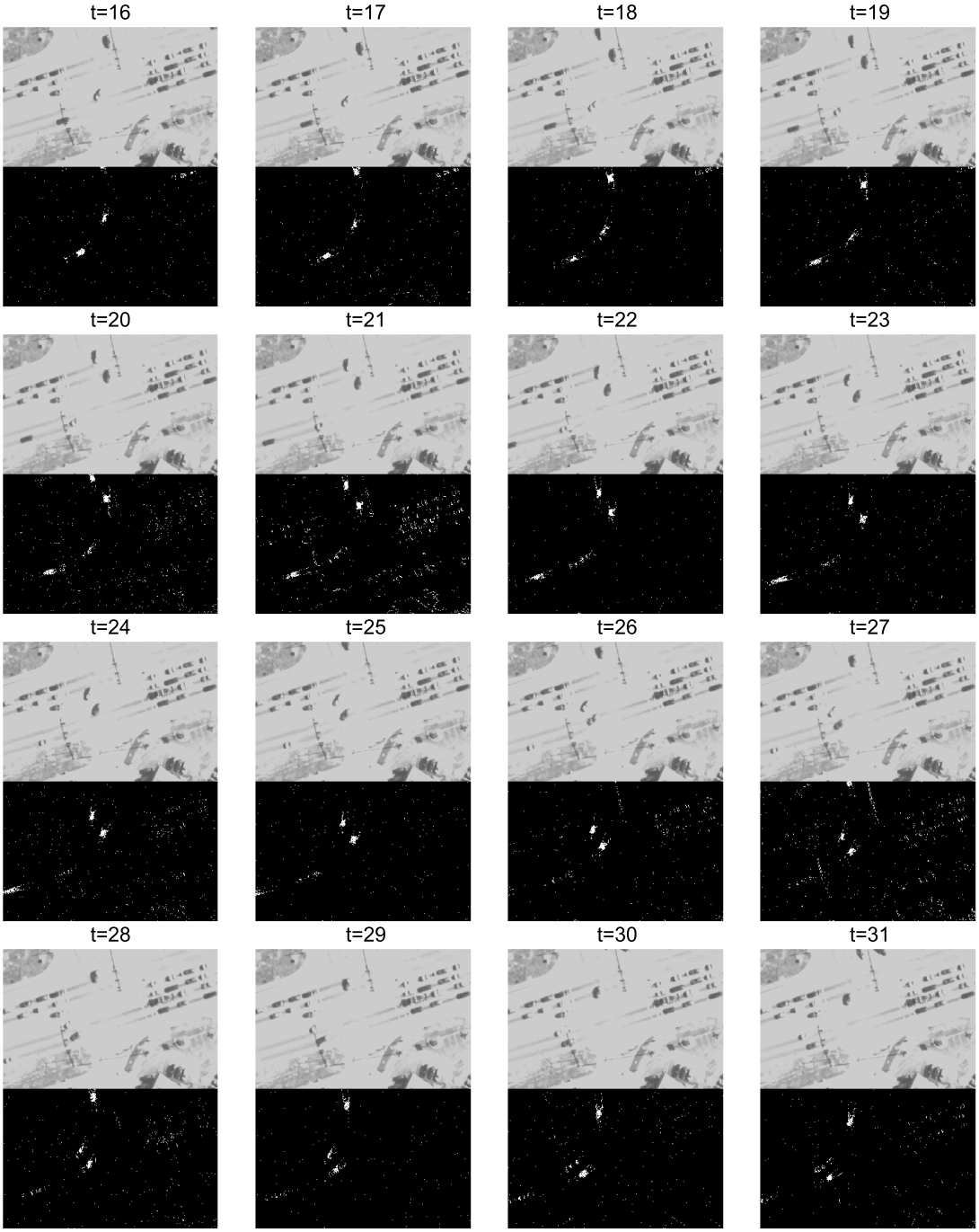


Figure 7.23: Video processed by the GF3 CPD chip (figure 2). For each time step, the top image is the greyscale image used as the input for the CPD processing algorithm.

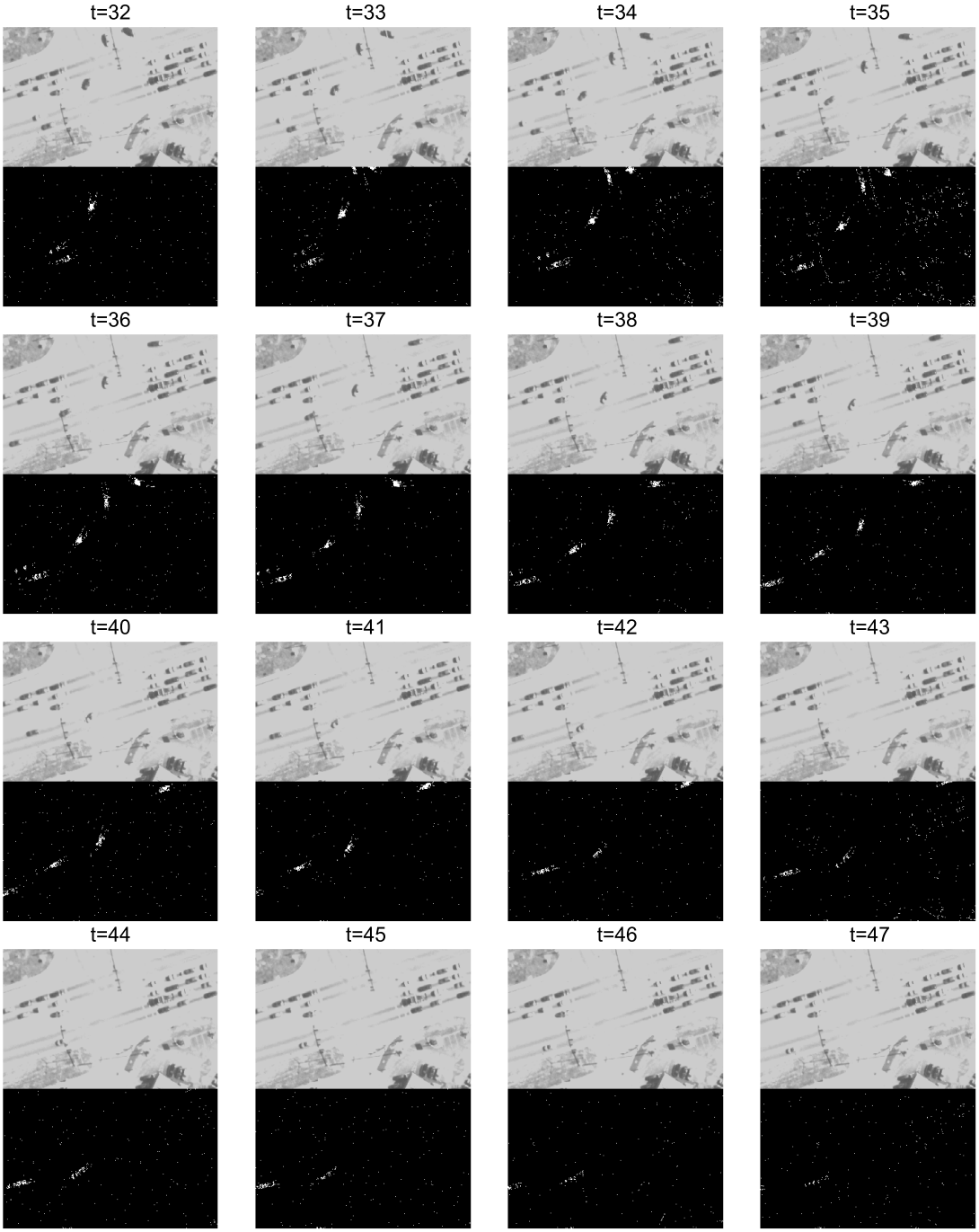


Figure 7.24: Video processed by the GF3 CPD chip (figure 3). For each time step, the top image is the greyscale image used as the input for the CPD processing algorithm.

Chapter 8

Design of a True Random Number Generator using RTN noise

8.1 Introduction

As seen in Chapter 7, and in also other processing units that will be shown later as well, random numbers are necessary when processing in the stochastic domain. For the case of the CPD processor presented in test chip GF3, the number of random numbers required for each core was 17, with a computational time of 4096 clock cycles. The reality is that, when performing decoding, the number of random numbers generators (RNGs) and the amount of time used in the processing do not need to be maintained for the same level of accuracy to be obtained. Actually one only needs to maintain constant the multiplication $N_{RN}.P_{time}$, where N_{RN} is the number of random

numbers and P_{time} the computational time. As an example, let's take a look at Figure 8.1. In this figure, a trade-off between frequency of operation and number of RNGs and silicon area is exemplified. Sequences $A = [A_1 A_2]$ and $B = [B_1 B_2]$. The first two streams A and B are running at frequency F, and the other four ones are running at $F/2$. For the last four streams one can see the silicon area and the number of RNGs running at half the speed is doubled. Results obtained by the *Decoder* blocks are exactly the same. This deserialization process can be conveniently linked to neural networks.[65] The brain is massively parallel in its processing, and, neuron-wise, frequencies of operation are in the KHz range. These low frequencies are only obtained if massive replication is applied, like in the example in Figure 8.1. If one could afford increasing parallelism and reduction of frequencies in the KHz range, ASICs could be designed to run with power supplies down to $100mV$,[37] reaching ultra-low power dissipation. This approach becomes very interesting for machine learning classifiers such as convolutional neural networks.[66]

Flipping a coin or drawing numbered marbles from a bag are some examples of how random numbers can be generated, but what if millions of random numbers per second are required? There is no viable approach that would allow these daily life examples achieve this task. It is for this reason that LFSRs are really useful, as they provide pseudo random numbers at very high rates. The problem with LFSRs is that they look random, but they are actually completely deterministic, and so if one required very large number of RNGs, at some point correlation between numbers

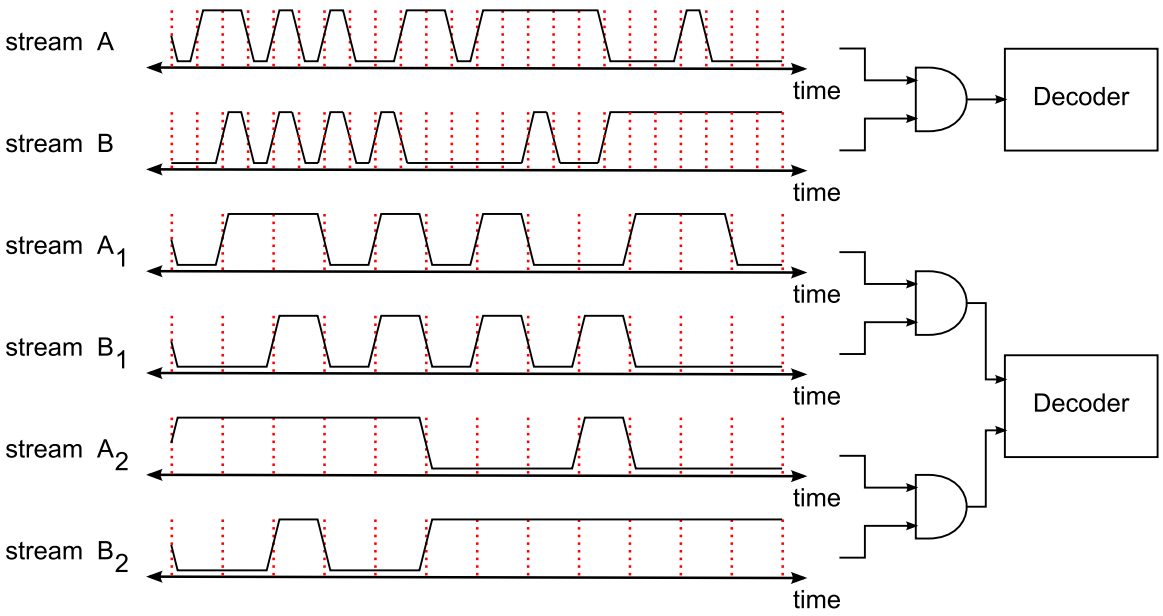


Figure 8.1: Tradeoff between computational time and silicon area. Streams A and B are processed by the AND gate at frequency F. Streams A_1 is ANDed with B_1, and streams A_2 is ANDed with B_2 at frequency $F/2$. $A = [A_1 A_2]$ and $B = [B_1 B_2]$. The decoder reaches the same answer, but for the second approach frequency is halved, with an increase to double in silicon area and the number of random number generators working at half the speed.

will show up. All the analysis done in Chapter 7 assumes the independence of all of the random number generators (RNG). If correlation between RNGs is found, then all the computing elements shown in Figure 7.3 would cease to work. It is for this reason that the generation of a true random number source is of most importance for processors working stochastically.

Many applications such as stochastic processors and random sampling ICs require the generation of large quantity of random numbers. In these cases, an ideal Bernoulli RNG source with probability $p = 0.5$ is required, but designing a true RNG source with this characteristic, while maintaining area efficiency, is a challenging problem.

CHAPTER 8. DESIGN OF A TRUE RANDOM NUMBER GENERATOR USING RTN NOISE

Randomness based on physical variations and imperfections of circuits is generally combined with other deterministic factors. True randomness requires the separation of these two sources and the elimination, or at least minimization, of the deterministic sources. One of the main deterministic sources of variation in integrated circuits is device mismatch. A technique that can reduce the impact of mismatch is the use of feedback techniques,[67–72] but the results rely strongly on a precise modeling of mismatch. When a feedback loop is implemented, the Bernoulli probability p used in the generation of the random numbers changes over time, and can be actually considered a random variable itself, namely P, from which samples are taken. Based on this observation, it was established that constraint that the true mean of the Bernoulli probability p produced by the feedback loop must be $E(P(n)) = 0.5$.

Three main approaches can be found in the literature for the design of true random number generators (TRNGs): direct amplification, oscillator sampling and discrete time chaos. Approach in[67] proposes the design of a TRNG that uses amplified thermal noise added to the output of an A/D-based discrete chaos generator that drives another oscillator sampled at a lower user-defined frequency; the system occupies a total chip area of $1.5mm^2$ in a $2\mu m$ process.[68] shows experimentally a TRNG based on the combination of thermal noise amplification and chaos, which occupies $0.21mm^2$ in a $65nm$ technology. More recently, several small circuits have been proposed, mainly driven by the need present in RFID ICs. On the other hand,[69] presents a meta-stable latch-based TRNG with a bias adjustment based on the decision time; the full circuit

was fabricated in a $0.13\mu m$ bulk CMOS technology with an area of $0.145mm^2$. In[70] a "DC-nulling" circuit is presented in a $0.35\mu m$ process that uses $0.031mm^2$ based on floating gates; a maximum operating frequency of $100Khz$ is claimed along with power estimates of $9.39\mu W$ @ $5kbps$. The approach presented in[7172] relies on a cross-coupled pair of inverters, with correction mechanisms. The circuit occupies $4004\mu m^2$ in a $45nm$ technology process with a power efficiency of $3\mu W/MHz$.

Unfortunately, no system model is provided in the previously mentioned work which justifies the design of the feedback loop. In this work, a novel design of a TRNG that achieves a true $E(P(n)) = 0.5$ is presented, something that none of the cited papers achieve. The architecture proposed in this work[73] is based on the perturbation of a Sigma-Delta modulator using random telegraph noise (RTN).[74–79]

8.2 Closed-Loop Controlled RNG

8.2.1 Architectural Description

In the work presented here, the change in current a small transistor suffers due to random telegraph noise (RTN) is exploited in the generation of a Bernoulli stochastic process with probability $p = 0.5$. Transistors are not fabricated perfectly and electrons can get trapped in their gate. The occupancy of these electron traps is random, randomly an electron gets trapped and randomly it is released, and this occupancy generates a change in the threshold voltage of the transistor. This is the RTN phe-

nomenon that will be used in the system to generate uniform random samples that will be presented in this work. The usage of a Sigma-Delta modulator[80–82] with an output randomly affected by RTN will help achieve this goal, and by introducing a DAC at the input of it, the probability of the generated random number will be controlled.

Stochastic processors usually require numbers to be encoded into stochastic streams of zeros and ones, where the Bernoulli probability of one p encodes the desired number, as seen in Figure 7.2. In doing this, a source of uniform random numbers is required, and for each drawn sample, if this value is lower than the value one wants to encode, a '1' is sent at the output of the encoder, otherwise a '0'. In this comparison, both the encoded number and the random source are represented as N-bit numbers. The probability distribution of the random numbers in the range 0 to 2^N-1 has to be uniform for the encoding process to work correctly. In an N-bit number, the probability of one for bit i will be p_i, and so Equation 8.1 will show the probability of an N-bit number, where b_i is the value of bit i, that can be either zero or one.

$$P_{N-bit} = \prod_{i=0}^{N-1} p_i^{b_i}(1-p_i)^{1-b_i} \tag{8.1}$$

If a N-bit random number source is desired to be distributed uniformly, then the probability of any of the 2^N numbers has to be $P_{N-bit} = 1/N$. For this to happen it can be shown that $p_i = 0.5$. The problem that needs to be faced is that even if

one designed a random number source to output a specific probability p, in reality mismatch in the fabrication process will make the desired value of p to deviate. Because of the very specific $p = 0.5$ desired probability, a digital analog converter (DAC) was decided to be placed at the input of a Sigma-Delta modulator so that the output probability value can be controlled.

The requirement of a vast quantity of random number sources for many applications prevented the option of performing the feedback control loop off-chip from moving forward. When thousands of random sources are necessary, individual tweaking for each source becomes an impossible task. The best option would be to come up with a design performing self-control automatically no matter what the mismatches encountered in the fabrication process are. The first system approach is shown in Figure 8.2, where the control is now done on-chip for each of the RNGs.

Enclosed by the dotted line the Sigma-Delta random number generator is defined. Here three blocks are identified, a gain A, an input *B(z)* and the *Encoder* block. The *Encoder* block translates a number into a stochastic stream as mentioned before. The *Decoder* block estimates the mean value of a stochastic stream, which in this case is the Bernoulli probability $P(n)$ at the input of the *Encoder* block. Here the Sigma-Delta is being modeled as a linear function, where its output value is $y = Ax + B$. In theory B should be zero, and then for a input *ctrl_i = 0*, the *Encoder* block would generate a probability value of $P(n) = 0.5$ at its output. The problem is that B could be not zero, and then one needs to correct for this value with the input *ctrl_i*.

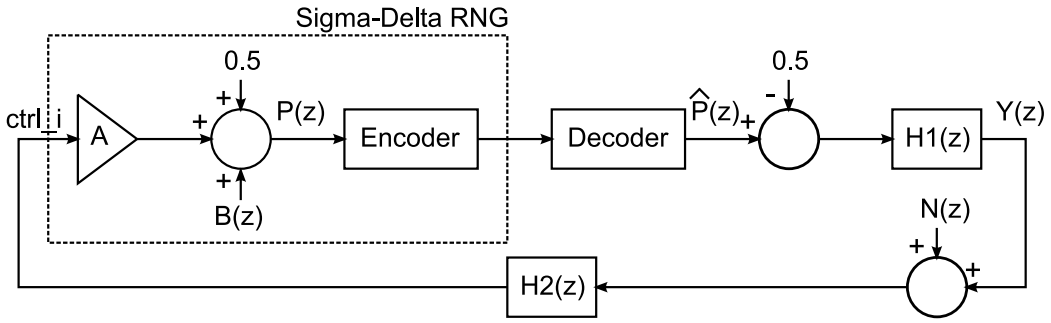


Figure 8.2: First approach to a feedback controlled Sigma-Delta based TRNG.

So that a transfer function can be obtained for the system, $B(z)$ is defined as a system input. The input $N(z)$ corresponds to the noise input for the system. In this case $H1(z) = z^{-M}$, meaning its just a M steps delay line, and $H2(z)$ is the control feedback. Considering $Y(z)$ the output of the system, the transfer function for the system in Figure 8.2 is extracted.

$$(Y(z)H2(z)A + B(z))z^{-M} = Y(z) \tag{8.2}$$

$$Y(z)(1 - z^{-M}AH2(z)) = B(z)z^{-M}$$

$$\Rightarrow \frac{Y(z)}{B(z)} = \frac{z^{-M}}{(1 - z^{-M}AH2(z))}$$

In order for the whole system to work, two different frequencies needed to be defined, F_{sys} and F_{rand}. Frequency F_{sys} defines the rate at which samples are generated at the output of the *Decoder* block. F_{rand} is the frequency at which the *Encoder* outputs random bits. The *Decoder* block will integrate the output coming from the *Encoder* for $F_{rand}/F_{sys} = K_{int}$ clock cycles, and with this decoding process, a noisy

decoded sample is generated. The output of the *Decoder* block $\hat{P}(n)$ is interpreted as a number $\in [0;1]$, then this noisy sample, following the Central Limit Theorem, is Gaussian distributed with $\sigma^2 = P(n)(1-P(n))/K_{int}$ and mean equal to $P(n)$. The value $P(n)$, as mentioned before, is the Bernoulli probability at the input of the *Encoder* block. This is where the first source of noise for the system is found, and this noise displays no mean due to the fact that the decoder estimator mean, is the true mean of the decoded stochastic process.

Independently of the values A and $B(z)$ have, the output $Y(z)$ needs to always converge to zero. This implies that the output of the *Decoder* block needs to converge to 0.5, meaning that the output of the *Encoder* block is a Bernoulli random variable with probability that converges to $p = 0.5$. The simplest way to satisfy this condition is making sure that $H2(z)$ has a pole in 1, so that this pole becomes a zero for the overall system. Let's then propose $H2(z) = L_0/(1-z^{-1})$, and then one can obtain the resulting expression in Equation 8.3. The value for AL_0 has to make the system stable and the poles as far as possible from the zero at 1, so that a fast convergence can be achieved.

$$\frac{Y(z)}{B(z)} = \frac{z^{-M}(1-z^{-1})}{(1-z^{-1}-z^{-M}AL_0)} \tag{8.3}$$

The number of bits that represent the input *ctrl_i* in the *Sigma-Delta RNG* block is very limited. The reason for this is that it is desired to keep the area for this random number generator architecture as small as possible, and high resolution *DACs* can take a considerable amount of area. With this reduction in the number of bits

from $log_2(K_{int})$ to K_{dac} (number of bits for the DAC) the second source of noise is introduced, the quantization noise.

If the input $N(z)$ is now considered, the noise transfer function can be extracted (considering $B(z) = 0$). The transfer function is presented in Equation 8.4. This transfer function presents a problem, it does not filter out the mean of the noise. One of the requirements for $H2(z)$ is to have a pole in 1, so that it could be translated into a zero in 1 for Equation 8.3. But in this case, for the noise transfer function 8.4, the poles in $H2(z)$ are not translated as zeros, and no modifications to $H2(z)$ will change this.

$$\frac{Y(z)}{N(z)} = \frac{H2(z)Az^{-M}}{(1 - H2(z)Az^{-M})} = \frac{L_0 Az^{-M}}{(1 - z^{-1} - z^{-M}AL_0)} \tag{8.4}$$

Using the superposition property of LTI systems, three cases are analyzed:

1. **Case 1** No noise present in the system, meaning $N(n) = 0$.

2. **Case 2** Only Gaussian noise is present in the system, meaning $B(n) = 0$ and $N(n) \sim \mathcal{N}(\mu = 0, \sigma^2 = P(n)(1 - P(n))/N_{int})$.

3. **Case 3** Only quantization noise is present in the system, meaning $B(n) = 0$ and $N(n) \sim Q(\mu \neq 0)$.

The transfer function for *ctrl_i(z)* with respect to *B(z)* is calculated:

$$\frac{ctrl\_i(z)}{B(z)} = \frac{z^{-M}L_0}{(1 - z^{-1} - z^{-M}AL_0)} \tag{8.5}$$

In **Case 1**, when the system settles, using the final value theorem for the Z-transform, the signal $ctrl\_i(n)$ will converge to the value:

$$\begin{aligned}\lim_{n\to+\infty} ctrl\_i(n) &= \lim_{z\to 1}(z-1)B(z)\frac{z^{-M}L_0}{(1-z^{-1}-z^{-M}AL_0)} \\ &= \lim_{z\to 1}(z-1)\frac{B}{1-z^{-1}}\frac{z^{-M}L_0}{(1-z^{-1}-z^{-M}AL_0)} = -\frac{B}{A}\end{aligned} \tag{8.6}$$

And this makes the *Encoder*'s input value $P(n) = 0.5$. For **Case 2** and **Case 3** one needs to calculate the expected value of *ctrl_i(n)* when $N(n)$ is a random input. Assuming that the random process $N(n)$ is stationary and each sample is independent of each other, and knowing that $E(y(n)) = H(z = 1)E(x(n))$, where $H = Y(z)/X(z)$ an LTI system:

$$\frac{ctrl\_i(z)}{N(z)} = \frac{L_0}{1-z^{-1}-AL_0z^{-M}} \Rightarrow E(ctrl\_i(n)) = -\frac{E(N(n))}{A} \tag{8.7}$$

It can be immediately seen that in **Case 2**, where the noise mean is zero, signal $ctrl\_i$ will have a mean value of zero as well. This makes the output probability of the *Encoder* not change when superposition of the first and second cases are considered. The situation is different for **Case 3**, where $E(N(n)) \neq 0$. If the quantization is done by truncating to the K_{dac} most significant bits, the mean for the quantization noise will not necessarily be zero. Consequently another way of performing a reduction from log_2K_{int} to K_{dac} bits precision needs to be considered. This is the reason why a log_2K_{int} bits to K_{dac} bits digital Sigma-Delta converter is introduced here for the

quantization process.

Figure 8.3 shows the updated design for the system in Figure 8.2, where the *Decoder* block has been updated with a one step delay which is necessary for the decoding process. The two sources of noise are shown in the system, where $GN(z)$ is the Gaussian noise from the decoding process, and $QN(z)$ is the quantization noise due to the digital Sigma-Delta. The transfer function shown in Equation 8.3 and the transfer function for the Gaussian noise do not change. Only the one corresponding to the quantization noise $QN(z)$ changes, which is shown in Equation 8.8. In this equation the mean of the noise is being rejected, which is exactly what is needed. The requirements on L_0A for making the system in Equation 8.7 stable are the same ones as before, and they are that $AL_0 \in [-1; 0]$. The gain L_0 has been moved all the way to the *Sigma-Delta RNG* input, this is because the value for L_0 will depend jointly with the value of A. If a division or multiplication had to be performed for L_0, this will take a significant amount of logic to perform it in digital, where on the other hand changing the gain in the *Sigma-Delta RNG* DAC will be much easier and will require less area.

$$\frac{Y(z)}{QN(z)} = \frac{z^{-1}(1-z^{-1})^2}{1-z^{-1}-L_0Az^{-2}} = -\frac{z^{-1}(1-z^{-1})^2}{AL_0(\frac{-1-\sqrt{1+4AL_0}}{2AL_0}-z^{-1})(\frac{-1+\sqrt{1+4AL_0}}{2AL_0}-z^{-1})} \quad (8.8)$$

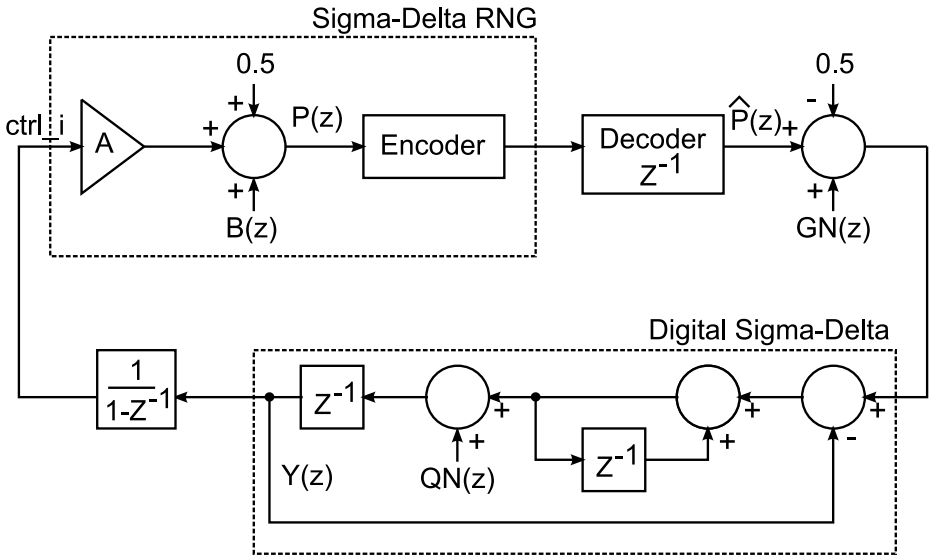


Figure 8.3: Second approach to a feedback controlled Sigma-Delta based TRNG.

8.2.2 Modeling the System Noise

Input $P(n)$ to the *Encoder* block is a random variable with a mean μ_P and variance σ_P^2. For the designed system $\mu_P = 0.5$. For every time in n, a new sample $p(n)$ is drawn from $P(n)$'s distribution. A probability density function is difficult to obtain for $P(n)$ and then the assumption will be that this distribution is normal with the before mentioned parameters μ_P and σ_P^2. The following equation shows the decoding estimator used in the system.

$$\hat{P}(\vec{X}) = \frac{1}{K_{int}} \sum_{i=1}^{K_{int}} X_i \tag{8.9}$$

In this equation $\vec{X} = [X_1, X_2, ..., X_{K_{int}}]$ is a vector of K_{int} random variables distributed according to $P(n)$'s distribution. When drawing a sample from the random

variable $P(n)$, this sample $p(n)$ will not necessarily be 0.5 for those K_{int} bits, but one knows that $P(n)$ distribution has a mean of 0.5 because of the quatization error mean rejection seen in Equation 8.8. The statistic in Equation 8.9 has its own distribution, and one can then proceed to calculate its mean $\mu_{\hat{P}}$ and variance $\sigma^2_{\hat{P}}$. From now on p_{val} will refer to probability value.

$$\begin{aligned} E_{\vec{X}}(\hat{P}(\vec{X})) &= \frac{1}{K_{int}}\sum_{i=1}^{K_{int}} E_{X_i}(X_i) = \frac{1}{K_{int}}\sum_{i=1}^{K_{int}}(0.p_{val}(X_i=0)+1.p_{val}(X_i=1)) \\ &= \frac{1}{K_{int}}\sum_{i=1}^{K_{int}} P(n) = P(n) = \mu_{\hat{P}} \qquad (8.10) \\ E_{\vec{X}}((\hat{P}(\vec{X})-\mu_{\hat{P}})^2) &= E_{\vec{X}}\left(\left(\frac{1}{K_{int}}\sum_{i=1}^{K_{int}} X_i - \mu_{\hat{P}}\right)^2\right) \\ &= \frac{1}{K_{int}^2}\sum_{i=1}^{K_{int}}\sum_{j=1}^{K_{int}} E_{X_i.X_j}((X_i-\mu_{\hat{P}})(X_j-\mu_{\hat{P}})) \\ &= \frac{1}{K_{int}^2}\left(\sum_{i=1}^{K_{int}} E_{X_i}((X_i-\mu_{\hat{P}})^2) + \sum_{\substack{i=1 \\ i\neq j}}^{K_{int}}\sum_{j=1}^{K_{int}} E_{X_i.X_j}((X_i-\mu_{\hat{P}})(X_j-\mu_{\hat{P}}))\right) \\ &= \frac{1}{K_{int}^2}\sum_{i=1}^{K_{int}} P(n)(1-P(n)) = \frac{1}{K_{int}}P(n)(1-P(n)) = \sigma^2_{\hat{P}} \qquad (8.11) \end{aligned}$$

Equation 8.10 confirms that the *Encoder* block plus the *Decoder* block can be thought as just a cable. Equation 8.11 additionally tells that it is not just a cable, but a noisy cable that adds a variance equal to $P(n)(1-P(n))/K_{int}$. If now one considers that $P(n)$ to be a random variable as well, from which samples $p(n)$ can be

taken, then one can calculate an average variance for the *Decoder* block:

$$E_{P(n)}\left(\frac{1}{K_{int}}P(n)(1-P(n))\right) = \frac{1}{K_{int}}(E_{P(n)}(P(n)) - E_{P(n)}(P(n)^2)) \\ = \frac{1}{K_{int}}(\mu_P(1-\mu_P) - \sigma_P^2) \tag{8.12}$$

If random variable $P(n)$ was not random but deterministic, meaning $P(n) \sim \delta(P(n) - \mu_P)\mu_P$, and $\sigma_P^2 = 0$ then the variance $E((\hat{P}(\vec{X}) - \mu_P)^2)$ will just be $\mu_P(1-\mu_P)/K_{int}$ as it is known from Bernoulli distributions. Equation 8.11 shows that the variance is inversely proportional to the number of samples chosen for the decoding process. This is one of the reasons why K_{int} is required to be a large number. It can now be concluded that the Gaussian noise added to the system is distributed $GN(n) \sim \mathcal{N}(0, \frac{1}{K_{int}}(\mu_P(1-\mu_P) - \sigma_P^2))$.

Now three different transfer functions are now defined for the signal $P(n)$ depending on the system input:

$$H_{P,B} = \frac{(1-z^{-1})}{(1-z^{-1}-z^{-2}AL_0)} \\ H_{P,GN} = \frac{AL_0z^{-1}}{(1-z^{-1}-z^{-2}AL_0)} \\ H_{P,QN} = \frac{AL_0z^{-1}(1-z^{-1})}{(1-z^{-1}-z^{-2}AL_0)} = 1 - \frac{1-(AL_0+1)z^{-1}}{(1-z^{-1}-z^{-2}AL_0)} \tag{8.13}$$

With these transfer functions one can define the auto-correlation function for

$P(n)$, where $*$ is the convolution operator:

$$\begin{aligned} R_{PP} = {} & h_{P,B}(n) * h_{P,B}(-n) * R_{BB} \\ & + h_{P,GN}(n) * h_{P,GN}(-n) * R_{GNGN} \\ & + h_{P,QN}(n) * h_{P,QN}(-n) * R_{QNQN} \end{aligned} \tag{8.14}$$

R_{BB}, R_{GNGN} and R_{QNQN} are the auto-correlation functions for inputs $B(n)$, Gaussian noise $GN(n)$ and quantization noise $QN(n)$. Input $B(n)$ is being considered a step function, for which its value will be deterministic, so no auto-correlation function can be defined for it, thus $R_{BB} = 0$. Gaussian noise $GN(n)$ will be considered not correlated, and then $R_{GNGN}(k) = \delta(k)\sigma_{GN}^2 = \delta(k)(\mu_P(1-\mu_P) - \sigma_P^2)/K_{int}$ due to Equation 8.12. Finally it is assumed that the quantization noise is not correlated, then $R_{QNQN}(k) = \delta(k)\sigma_{QN}^2$. Finally:

$$R_{PP} = h_{P,GN}(n) * h_{P,GN}(-n)\frac{1}{K_{int}}(\mu_P(1-\mu_P) - \sigma_P^2) + h_{P,QN}(n) * h_{P,QN}(-n)\sigma_{QN}^2 \tag{8.15}$$

An expression for the impulse responses for the different transfer functions need to be found now. Assuming a generic transfer function $H(z)$:

$$H(z) = \frac{b_1 z^{-1} + b_0}{a_2 z^{-2} + a_1 z^{-1} + a_0} \tag{8.16}$$

Which can be decomposed in the following way:

$$H(z) = \frac{G1}{\left(1 - \frac{2a_2}{(-a_1+\sqrt{a_1^2-4a_2a_0})} z^{-1}\right)} + \frac{G2}{\left(1 - \frac{2a_2}{(-a_1-\sqrt{a_1^2-4a_2a_0})} z^{-1}\right)} \tag{8.17}$$

Where $G1$ and $G2$ are:

$$\begin{aligned} G1 &= b_0 \frac{a_1 + \sqrt{a_1^2 - 4a_2a_0}}{2\sqrt{a_1^2 - 4a_2a_0}} + b_1 \frac{(a_1 + \sqrt{a_1^2 - 4a_2a_0})(-a_1 + \sqrt{a_1^2 - 4a_2a_0})}{4a_2\sqrt{a_1^2 - 4a_2a_0}} \\ G2 &= b_0 \frac{-a_1 + \sqrt{a_1^2 - 4a_2a_0}}{2\sqrt{a_1^2 - 4a_2a_0}} - b_1 \frac{(a_1 + \sqrt{a_1^2 - 4a_2a_0})(-a_1 + \sqrt{a_1^2 - 4a_2a_0})}{4a_2\sqrt{a_1^2 - 4a_2a_0}} \end{aligned} \tag{8.18}$$

By knowing now that:

$$\sum_{n=0}^{+\infty} A^n z^{-n} = \frac{1}{1 - Az^{-1}} \tag{8.19}$$

One now needs to calculate the impulse response $h_{P,GN}(n)$ and $h_{P,QN}(n)$ if the auto-correlation and variance for $P(n)$ is desired to be calculated. Here $\delta(n)$ is the unitary impulse, and $u(n)$ the unitary step, and $\Gamma = \sqrt{1 + 4AL_0}$:

$$\begin{aligned} h_{p,GN}(n) &= u(n) \frac{(1+\Gamma)(1-\Gamma)}{4\Gamma} \left(\left(\frac{-2AL_0}{1+\Gamma} \right)^n - \left(\frac{-2AL_0}{1-\Gamma} \right)^n \right) \\ h_{ctrl\_i,QN}(n) &= \delta(n) - u(n) \frac{(-1+\Gamma)(3AL_0 + 1 + \Gamma(AL_0+1))}{4\Gamma AL_0} \left(\frac{-2AL_0}{1+\Gamma} \right)^n \\ &\quad - u(n) \frac{(1+\Gamma)(3AL_0 + 1 - \Gamma(AL_0+1))}{4\Gamma AL_0} \left(\frac{-2AL_0}{1-\Gamma} \right)^n \end{aligned} \tag{8.20}$$

One can now calculate the whole correlation function by using 8.15 and 8.20. What is more important right now is to obtain an expression for the variance of

$P(n)$.

$$\begin{aligned}
\sigma_P^2 &= \frac{1}{K_{int}}(\mu_P(1-\mu_P)-\sigma_P^2)\sum_{\forall n} h_{P,GN}^2(n) + \sigma_{QN}^2 \sum_{\forall n} h_{P,QN}(n)^2 \\
\sigma_P^2\left(1+\frac{1}{K_{int}}\sum_{\forall n} h_{P,GN}^2(n)\right) &= \frac{1}{K_{int}}\mu_P(1-\mu_P)\sum_{\forall n} h_{P,GN}^2(n) + \sigma_{QN}^2 \sum_{\forall n} h_{P,QN}(n)^2 \\
\sigma_P^2 &= \frac{\mu_P(1-\mu_P)\sum_{\forall n} h_{P,GN}^2(n) + K_{int}\sigma_{QN}^2\sum_{\forall n} h_{P,QN}(n)^2}{\left(K_{int}+\sum_{\forall n} h_{P,GN}^2(n)\right)}
\end{aligned} \tag{8.21}$$

$$\begin{aligned}
\sum_{\forall n} h_{p,GN}^2(n) =& \frac{(1+\Gamma)^2(1-\Gamma)^2}{(4\Gamma)^2}\left(\frac{(1+\Gamma)^2}{(1+\Gamma)^2-4(AL_0)^2}+\frac{(1-\Gamma)^2}{(1-\Gamma)^2-4(AL_0)^2}\right. \\
&\left. -2\frac{(1+\Gamma)(1-\Gamma)}{(1+\Gamma)(1-\Gamma)-4(AL_0)^2}\right) \\
\sum_{\forall n} h_{p,QN}^2(n) =& -1+\frac{(1+\Gamma)^2(1-\Gamma)^2}{(4\Gamma AL_0)^2}\left(\frac{(3AL_0+1+\Gamma(AL_0+1))^2}{(1+\Gamma)^2-4(AL_0)^2}\right. \\
&+\frac{(3AL_0+1-\Gamma(AL_0+1))^2}{(1-\Gamma)^2-4(AL_0)^2} \\
&\left. -2\frac{(3AL_0+1+\Gamma(AL_0+1))(3AL_0+1-\Gamma(AL_0+1))}{(1+\Gamma)(1-\Gamma)-4(AL_0)^2}\right)
\end{aligned}$$

For the signal $P(n)$, in Equation 8.21, an expression for variance σ_P^2 was found as a function of the quantization noise variance σ_{QN}^2. Simulations were run using *Matlab*'s *Simulink* varying the quantization bits K_{dac} from 2 to 6, integration time K_{int} from 2^6 to 2^{13}, and four different values for the gain AL_0 -0.1, -0.2, -0.4 and -0.8. For each of the simulations runs, once the system stabilizes, 1024 samples were

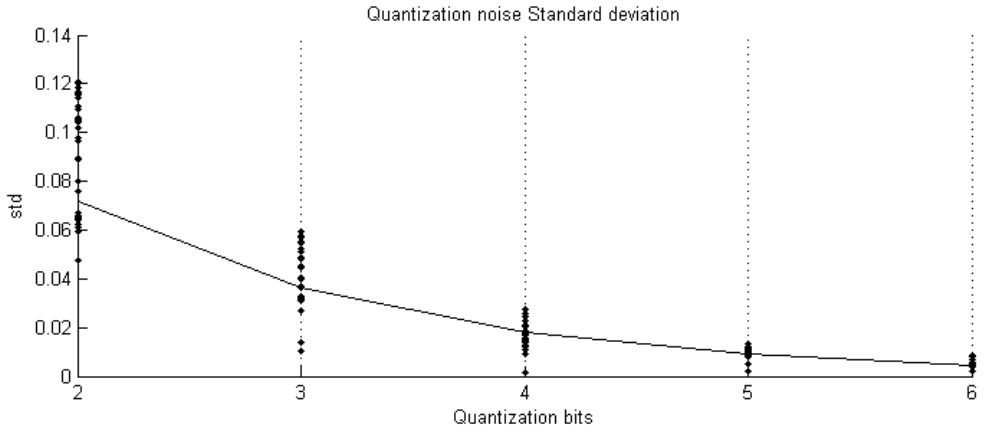


Figure 8.4: $P(n)$ standard deviation for when quantization noise is uniform. The solid line corresponds to the standard deviation for the case quantization noise is uniform. For each of the quantization bits values K_{dac}, simulations were run changing integration times K_{int} and gain AL_0 and estimations of σ_{QN} were found as the dots.

used to calculate σ_{QN}. Figure 8.4 shows the results found from these simulations that were used to estimate the quantization noise σ_{QN}. Quantization noise is represented with the variable $QN(n)$. The difference between the maximum and minimum value this variable can take is $1/2^{K_{dac}}$. In assuming $QN(n)$ as uniform, then its standard deviation would be $1/\sqrt{2^{2K_{dac}}12}$. In Figure 8.4, the points joined with lines correspond to the standard deviation for the case the distribution of $QN(n)$ was uniform. With these results, it is reasonable to assume that quantization noise is uniform, and then, from now on, the variance σ^2_{QN} will be changed to $1/2^{2K_{dac}}12$.

Considering uniformity in quantization noise $QN(n)$, Tables 8.1, 8.2, 8.3 and 8.4 present the different values for the standard deviation σ_P for the variation of gain AL_0, integration time K_{int} and quantization bits K_{dac}. In these tables, as K_{int} and K_{dac} increase, σ_P decreases as it is expected. Additionally, one can see that if one

K_{int} \ K_{dac}	2	3	4	5	6
64	0.1453	0.1119	0.1018	0.0992	0.0985
128	0.1288	0.0885	0.0752	0.0715	0.0705
256	0.1195	0.0737	0.0568	0.0517	0.0503
512	0.1144	0.0649	0.0446	0.0379	0.0360
1024	0.1118	0.0600	0.0370	0.0285	0.0259
2048	0.1105	0.0573	0.0325	0.0223	0.0190
4096	0.1098	0.0560	0.0300	0.0185	0.0143
8192	0.1095	0.0553	0.0287	0.0163	0.0112

Table 8.1: Standard deviation σ_P calculation. Calculation of σ_P for $P(n)$ for the case $AL_0 = -0.8$, varying K_{dac} and K_{int}.

K_{int} \ K_{dac}	2	3	4	5	6
64	0.0516	0.0424	0.0398	0.0391	0.0389
128	0.0437	0.0323	0.0288	0.0278	0.0276
256	0.0392	0.0258	0.0212	0.0199	0.0196
512	0.0367	0.0219	0.0162	0.0144	0.0139
1024	0.0354	0.0196	0.0129	0.0106	0.0100
2048	0.0347	0.0184	0.0109	0.0081	0.0072
4096	0.0344	0.0177	0.0098	0.0065	0.0053
8192	0.0342	0.0174	0.0092	0.0055	0.0040

Table 8.2: Standard deviation σ_P calculation. Calculation of σ_P for $P(n)$ for the case $AL_0 = -0.4$, varying K_{dac} and K_{int}.

already has a value for K_{dac} in mind, the reduction in σ_P as one increases K_{int} has an asymptotic behavior, and then increasing K_{int} might not be that beneficial when considering the amount of area that will be used to synthesize the digital blocks of the system.

K_{int} \ K_{dac}	2	3	4	5	6
64	0.0277	0.0243	0.0234	0.0231	0.0231
128	0.0224	0.0180	0.0168	0.0164	0.0163
256	0.0192	0.0139	0.0122	0.0117	0.0116
512	0.0174	0.0112	0.0090	0.0084	0.0082
1024	0.0164	0.0096	0.0069	0.0061	0.0058
2048	0.0159	0.0087	0.0056	0.0045	0.0042
4096	0.0157	0.0082	0.0048	0.0035	0.0030
8192	0.0155	0.0080	0.0044	0.0028	0.0023

Table 8.3: Standard deviation σ_P calculation. Calculation of σ_P for $P(n)$ for the case $AL_0 = -0.2$, varying K_{dac} and K_{int}.

K_{int} \ K_{dac}	2	3	4	5	6
64	0.0168	0.0155	0.0152	0.0151	0.0151
128	0.0130	0.0113	0.0108	0.0107	0.0107
256	0.0106	0.0084	0.0078	0.0076	0.0076
512	0.0091	0.0065	0.0056	0.0054	0.0054
1024	0.0083	0.0053	0.0042	0.0039	0.0038
2048	0.0079	0.0046	0.0032	0.0028	0.0027
4096	0.0077	0.0042	0.0026	0.0021	0.0019
8192	0.0075	0.0039	0.0023	0.0016	0.0014

Table 8.4: Standard deviation σ_P calculation. Calculation of σ_P for $P(n)$ for the case $AL_0 = -0.1$, varying K_{dac} and K_{int}.

8.2.3 Considerations for this TRNG

The focus will now be shifted to the stochastic processing units that use these random number generators. As mentioned before, $P(n)$ can be considered a random variable distributed as $P(n) \sim \mathcal{N}(\mu_P = 0.5, \sigma^2 = \sigma_P^2)$. A sample from this distribution will be generated every K_{int} samples at F_{rand} frequency. This behavior can be observed in Figure 8.5. The sampled probability $p(n)$ will be fixed for K_{int} samples

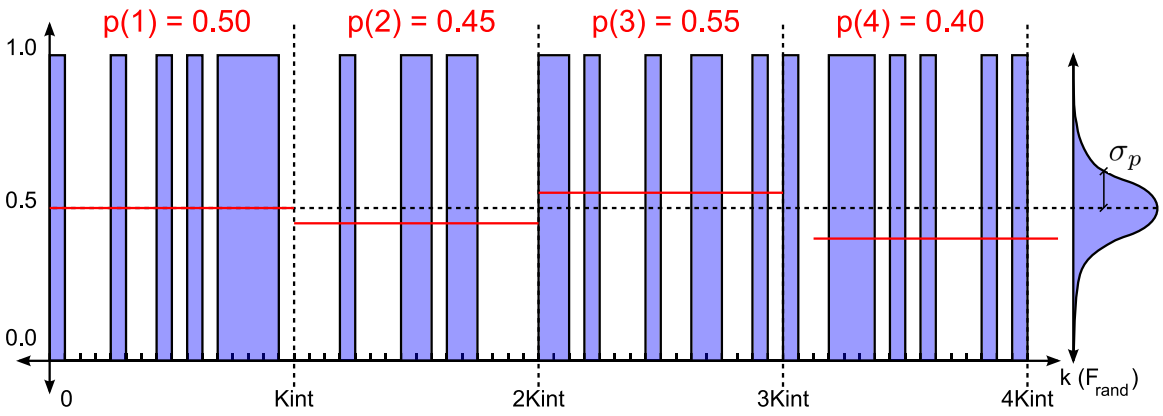


Figure 8.5: Encoder's output. The probability $p(n)$ is drawn from the distribution $P(n) \sim \mathcal{N}(\mu_P = 0.5, \sigma^2 = \sigma_P^2)$ every K_{int} samples at F_{rand} frequency.

at frequency F_{rand}. This setup might be a problem for applications that require a processing time $Proc_{time}$ close to K_{int}/F_{rand} because the encoded probability $p(n)$ might not be 0.5 for a particular time slot. If, on the other hand, applications where $Proc_{time} = K.K_{int}/F_{rand}$ with $K \in \mathbb{N}$, and $K >> 1$, then this problem is averaged out. On the side of the stochastic processing unit, if the streamed probability value were to be estimated, then the following estimator in Equation 8.22 can be presented.

$$\hat{P}_K = \frac{1}{KK_{int}} \sum_{i=1}^{K} \sum_{j=1}^{K_{int}} b_{ij}, b_{ij} \in \{0; 1\} \tag{8.22}$$

The mean and variance for the estimator $\hat{P}_K$ are now calculated. In their derivation, the identity $E((X - \mu_x)^2) = E(X^2) - \mu_x^2$, where X is a random variable, is used.

$$E(\hat{P_K}) = \frac{1}{K} \sum_{i=1}^{K} E_{P(i)} \left(\frac{1}{K_{int}} \sum_{j=1}^{K_{int}} E_b(b_{ij}) \right) = \frac{1}{K} \sum_{i=1}^{K} E_{P(i)}(P(i)) = \mu_P = 0.5 \tag{8.23}$$

$$
\begin{aligned}
E_{bP}((\hat{P_K} - \mu_P)^2) &= E_{bP}\left(\left(\frac{1}{KK_{int}}\sum_{i=1}^{K}\sum_{j=1}^{K_{int}} b_{ij} - \mu_P\right)^2\right) \qquad (8.24)\\
&= E_{bP}\left(\left(\frac{1}{KK_{int}}\sum_{i_1=1}^{K}\sum_{j_1=1}^{K_{int}} b_{i_1j_1} - \mu_P\right)\left(\frac{1}{KK_{int}}\sum_{i_2=1}^{K}\sum_{j_2=1}^{K_{int}} b_{i_2j_2} - \mu_P\right)\right)\\
&= E_{bP}\left(\frac{1}{K^2K_{int}^2}\sum_{i_1=1}^{K}\sum_{j_1=1}^{K_{int}}\sum_{i_2=1}^{K}\sum_{j_2=1}^{K_{int}} b_{i_1j_1}b_{i_2j_2} - \frac{2\mu_P}{KK_{int}}\sum_{i=1}^{K}\sum_{j=1}^{K_{int}} b_{ij} + \mu_P^2\right)\\
&= \frac{1}{K^2K_{int}^2}\sum_{i_1=1}^{K}\sum_{j_1=1}^{K_{int}}\sum_{i_2=1}^{K}\sum_{j_2=1}^{K_{int}} E_{bP}(b_{i_1j_1}b_{i_2j_2}) - \frac{2\mu_P}{KK_{int}}\sum_{i=1}^{K}\sum_{j=1}^{K_{int}} E_{bP}(b_{ij}) + \mu_P^2\\
&= \frac{1}{K^2K_{int}^2}\sum_{i_1=1}^{K}\sum_{j_1=1}^{K_{int}}\sum_{i_2=1}^{K}\sum_{j_2=1}^{K_{int}} E_{bP}(b_{i_1j_1}b_{i_2j_2}) - \mu_P^2
\end{aligned}
$$

The quadruple summation in the last term of Equation 8.24 is now expanded.

$$
\begin{aligned}
&\sum_{i_1=1}^{K}\sum_{j_1=1}^{K_{int}}\sum_{i_2=1}^{K}\sum_{j_2=1}^{K_{int}} E_{bP}(b_{i_1j_1}b_{i_2j_2}) = \sum_{i=1}^{K}\sum_{j=1}^{K_{int}} E_{bP}(b_{ij}^2) + \sum_{\substack{i_1=1\\[i_1,j_1]\neq[i_2,j_2]}}^{K}\sum_{i_2=1}^{K}\sum_{j_1=1}^{K_{int}}\sum_{j_2=1}^{K_{int}} E_{bP}(b_{i_1j_1}b_{i_2j_2})\\
&= \sum_{i=1}^{K}\sum_{j=1}^{K_{int}} E_{bP}(b_{ij}^2) + \sum_{i=1}^{K}\sum_{\substack{j_1=1\\j_1\neq j_2}}^{K_{int}}\sum_{j_2=1}^{K_{int}} E_{bP}(b_{ij_1}b_{ij_2})\\
&+ \sum_{i_1=1}^{K}\sum_{\substack{i_2=1\\i_1\neq i_2}}^{K}\sum_{j=1}^{K_{int}} E_{bP}(b_{i_1j}b_{i_2j}) + \sum_{i_1=1}^{K}\sum_{\substack{i_2=1\\i_1\neq i_2}}^{K}\sum_{j_1=1}^{K_{int}}\sum_{j_2=1}^{K_{int}} E_{bP}(b_{i_1j_1}b_{i_2j_2})\\
&= \sum_{i=1}^{K}\sum_{j=1}^{K_{int}} E_P(E_b((b_{ij} - P(i))^2 + P(i)^2) + \sum_{i=1}^{K}\sum_{\substack{j_1=1\\j_1\neq j_2}}^{K_{int}}\sum_{j_2=1}^{K_{int}} E_P(P(i)^2)\\
&+ \sum_{i_1=1}^{K}\sum_{\substack{i_2=1\\i_1\neq i_2}}^{K}\sum_{j=1}^{K_{int}} E_P(P(i_1)P(i_2)) + \sum_{i_1=1}^{K}\sum_{\substack{i_2=1\\i_1\neq i_2}}^{K}\sum_{j_1=1}^{K_{int}}\sum_{j_2=1}^{K_{int}} E_P(P(i_1)P(i_2))\\
&= \sum_{i=1}^{K}\sum_{j=1}^{K_{int}} E_P(P(i)(1 - P(i)) + P(i)^2) + \sum_{i=1}^{K}\sum_{\substack{j_1=1\\j_1\neq j_2}}^{K_{int}}\sum_{j_2=1}^{K_{int}} (\sigma_P^2 + \mu_P)
\end{aligned}
$$

$$
\begin{aligned}
&+\sum_{i_1=1}^{K}\sum_{\substack{i_2=1\\ i_1\neq i_2}}^{K}\sum_{j=1}^{K_{int}}\mu_P^2+\sum_{\substack{i_1=1\\ i_1\neq i_2,j_1\neq j_2}}^{K}\sum_{i_2=1}^{K}\sum_{j_1=1}^{K_{int}}\sum_{j_2=1}^{K_{int}}\mu_P^2\\
&=\sum_{i=1}^{K}\sum_{j=1}^{K_{int}}E_P(P(i))+\sum_{i=1}^{K}\sum_{\substack{j_1=1\\ j_1\neq j_2}}^{K_{int}}\sum_{j_2=1}^{K_{int}}(\sigma_P^2+\mu_P)\\
&+\sum_{i_1=1}^{K}\sum_{\substack{i_2=1\\ i_1\neq i_2}}^{K}\sum_{j=1}^{K_{int}}\mu_P^2+\sum_{\substack{i_1=1\\ i_1\neq i_2,j_1\neq j_2}}^{K}\sum_{i_2=1}^{K}\sum_{j_1=1}^{K_{int}}\sum_{j_2=1}^{K_{int}}\mu_P^2\\
&=KK_{int}\mu_P+(K^2K_{int}^2-KK_{int})\mu_P^2+KK_{int}(K_{int}-1)\sigma_P^2 \qquad (8.25)
\end{aligned}
$$

The last term in Equation 8.25 is used in the last term in Equation 8.24, then:

$$E_{bP}((\hat{P_K}-\mu_P)^2)=\frac{\mu_P(1-\mu_P)}{KK_{int}}+\frac{(K_{int}-1)\sigma_P^2}{KK_{int}} \qquad (8.26)$$

For a random number generator that perceives no variance in the value of $P(n)$, when using the mean estimator like in Equation 8.23, the variance in that estimator using N samples will be $\mu_P(1-\mu_P)/N$. For the case of this random number generator, N is equal to $K.K_{int}$, but since the drawn probability $p(n)$ is fixed over K_{int} samples, the expression in 8.26 deviates from $\mu_P(1-\mu_P)/N$. One cannot use a small number for K_{int} because of the need to use a large number of samples to decode the probability value set by the *Encoder* block. On the other hand, if one did $K_{int}=1$, for which the drawn probability $p(n)$ does not stay fixed for more than a sample, then one can observe that the expression in Equation 8.26 falls into the one for a ideal random number generator.

In solving the fact that a sample for the random variable $P(n)$ remains fixed for K_{int} time slots at frequency F_{rand}, a few approaches can be taken. One could always take only one sample from every K_{int} samples generated at the output of the encoder block. This approach would solve this problem, but the random numbers will be generated at the F_{sys} frequency which may be too slow for some applications. Another approach can be the one presented in Figure 8.6. In this figure N random number generators have been placed along with a NxN matrix of 1-bit registers and N N-input multiplexers. The multiplexers' control signal in this case are free running 0 to $(N-1)$ counter. This architecture will generate N 1-bit random number streams, but the multiplexers will additionally scramble the outputs from the RNG blocks. If this scenario is thought in terms of the first scenario presented in Equation 8.22, then the number K and K_{int} will be updated with KN and K_{int}/N. The statistic in Equation 8.22 will have the same expression, the mean will be equal, but the variance will be updated with the following expression:

$$E_{bP}((\hat{P_K}-\mu_P)^2) = \frac{\mu_P(1-\mu_P)}{KK_{int}} + \frac{(K_{int}/N-1)\sigma_P^2}{KK_{int}} \underset{K_{int}>>1}{\Rightarrow} \frac{\mu_P(1-\mu_P)}{KK_{int}} + \frac{\sigma_P^2}{KN} \quad (8.27)$$

In the case of Figure 8.6, the probability $P(n)$ translated to the outputs rng_1, ..., rng_N will present the same mean as before, $\mu_P = 0.5$. This is because bits are being sampled randomly in an uniform way from N RNGs, and each of these individual means will be $\mu_P = 0.5$ as well.

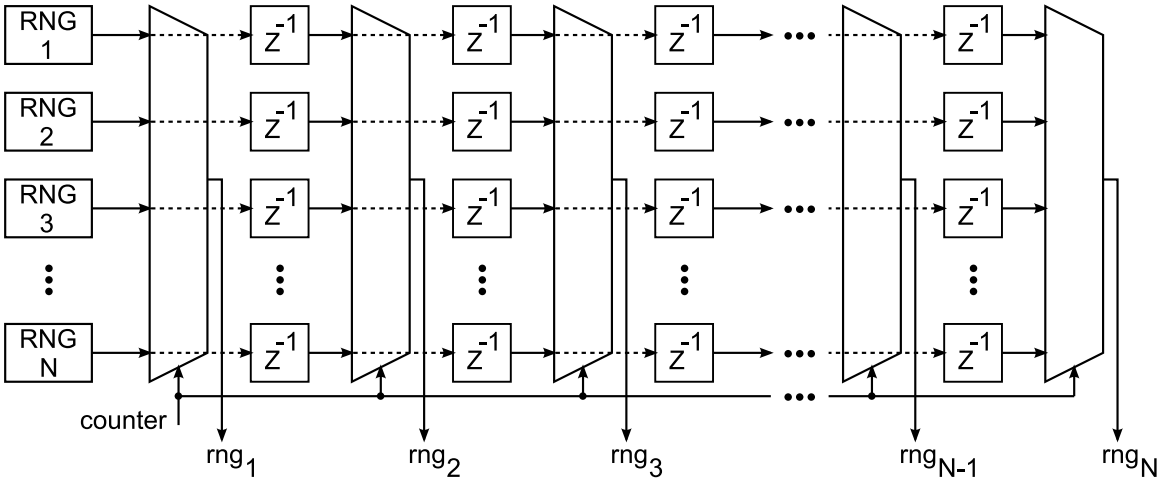


Figure 8.6: Scrambling architecture for the outputs of N random number generators.

The gain in this scrambling process is in time correlation. Let's calculate the correlation of two bits at the output of any of the RNG blocks from Figure 8.6, distanced by z time steps at F_{rand}, where $z \in [0, ..., K_{int}]$. For calculating the correlation $E((B(n+z) - \mu_P)(B(n) - \mu_P))$ where $B(n)$ and $B(n+z)$ are the two random variable mentioned bits, one needs to have an expression for the probability distribution $P(B(n), B(n+z))$. For obtaining this distribution one can first calculate the distribution $P(B(n), B(n+z), P(n) = p(n), P(n+z) = p(n+z))$, where in this case $p(n)$ and $p(n+z)$ are the probability of one for each of the samples from random variables $B(n)$ and $B(n+z)$, and marginalization over $P(n)$ and $P(n+z)$ will provide the

distribution one is looking for. Then:

$$\begin{aligned} & p_{val}(B(n), B(n+z), p(n), p(n+z)) \\ & = p_{val}(B(n+z)|p(n), p(n+z), B(n)).p_{val}(p(n), p(n+z), B(n)) \\ & = p_{val}(B(n+z)|p(n+z)).P_{val}(B(n)|p(n+z), p(n)) \\ & = p_{val}(B(n+z)|p(n+z)).P_{val}(B(n)|p(n)).p_{val}(p(n+z)|p(n)).p_{val}(p(n)) \end{aligned} \tag{8.28}$$

An expression for each of the multiplying terms in Equation 8.28 is found now:

$$p_{val}(B(n+z) = 0|P(n+z) = p(n+z)) = p(n+z)$$

$$p_{val}(B(n+z) = 1|P(n+z) = p(n+z)) = 1 - p(n+z)$$

$$\begin{aligned} \Rightarrow & p_{val}(B(n+z) = b(n+z)|P(n+z) = p(n+z)) \\ & = p(n+z)^{b(n+z)}(1 - p(n+z))^{1-b(n+z)} \end{aligned} \tag{8.29}$$

$$\Rightarrow p_{val}(B(n) = b(n)|P(n) = p(n)) = p(n)^{b(n)}(1 - p(n))^{1-b(n)} \tag{8.30}$$

$$P(n) \sim \mathcal{N}(\mu = \mu_P, \sigma_P^2) \tag{8.31}$$

$$p_{val}(P(n+z) = p(n+z)|P(n) = p(n)) =$$
$$\frac{K_{int} - z}{K_{int}}\delta(p(n+z) - p(n)) + \frac{z}{K_{int}}\mathcal{N}(p(n+z), \mu = \mu_P, \sigma_p^2) \tag{8.32}$$

In Equation 8.32, every K_{int} samples the underlying probability $p(n)$ changes, so with a probability $(K_{int} - z)/K_{int}$ one can say that probability $p(n)$ doesn't change. In order to make the equations not that long, the assumption that all the normal distributions $\mathcal{N}$ have a mean μ_P and variance σ_P^2 will be taken. Additionally all the values corresponding to n will be replaced with an subscript 1 and the ones from time $n + z$ a subscript 2. One can now proceed to calculate the joint distribution:

$$p_{val}(b_1, b_2) \tag{8.33}$$
$$= \int_{p_1}\int_{p_2} p_2^{b_2}(1-p_2)^{1-b_2}p_1^{b_1}(1-p_1)^{1-b_1}\mathcal{N}(p_1)\left(\frac{K_{int}-z}{K_{int}}\delta(p_2-p_1) + \frac{z}{K_{int}}\mathcal{N}(p_2)\right)dp_1dp_2$$
$$= \int_{p_1} p_1^{b_1}(1-p_1)^{1-b_1}\mathcal{N}(p_1)\left(\int_{p_2} p_2^{b_2}(1-p_2)^{1-b_2}\frac{K_{int}-z}{K_{int}}\delta(p_2-p_1)dp_2\right)dp_1$$
$$+ \int_{p_1} p_1^{b_1}(1-p_1)^{1-b_1}\mathcal{N}(p_1)\left(\int_{p_2} p_2^{b_2}(1-p_2)^{1-b_2}\frac{z}{K_{int}}\mathcal{N}(p_2)dp_2\right)dp_1$$
$$= \int_{p_1} p_1^{b_1}(1-p_1)^{1-b_1}\mathcal{N}(p_1)\frac{K_{int}-z}{K_{int}}p_1^{b_2}(1-p_1)^{1-b_2}dp_1$$
$$+ \int_{p_1} p_1^{b_1}(1-p_1)^{1-b_1}\mathcal{N}(p_1)\frac{z}{K_{int}}\left(\begin{cases}\int_{p_2}(1-p_2)\mathcal{N}(p_2)dp_2, & \text{if } b_2 = 0\\ \int_{p_2} p_2\mathcal{N}(p_2)dp_2, & \text{if } b_2 = 1\end{cases}\right)dp_1$$
$$= \frac{K_{int}-z}{K_{int}}\int_{p_1} p_1^{b_1+b_2}(1-p_1)^{2-b_1-b_2}\mathcal{N}(p_1)dp_1$$
$$+ \frac{z}{K_{int}}\int_{p_1} p_1^{b_1}(1-p_1)^{1-b_1}\mathcal{N}(p_1)((1-\mu_P)(1-b_2) + \mu_P b_2)dp_1$$

The combinations of all of the values for b_1 and b_2 need to be considered. In doing that, one obtains the joint distribution $p_{val}(b_1, b_2)$.

$$
\begin{aligned}
b_1 = 0, b_2 = 0 \Rightarrow & \frac{K_{int} - z}{K_{int}} \int_{p_1} (1 - p_1)^2 \mathcal{N}(p_1) dp_1 + \frac{z}{K_{int}} \int_{p_1} (1 - p_1) \mathcal{N}(p_1)(1 - \mu_P) dp_1 \\
= & \frac{K_{int} - z}{K_{int}} (\sigma_P^2 + (\mu_P - 1)^2) + \frac{z}{K_{int}} (1 - \mu_P)^2 \\
= & \frac{K_{int} - z}{K_{int}} \sigma_P^2 + (\mu_P - 1)^2 \\
b_1 = 0, b_2 = 1 \Rightarrow & \frac{K_{int} - z}{K_{int}} \int_{p_1} p_1 (1 - p_1) \mathcal{N}(p_1) dp_1 + \frac{z}{K_{int}} \int_{p_1} (1 - p_1) \mathcal{N}(p_1) \mu_P dp_1 \\
= & \frac{K_{int} - z}{K_{int}} (\mu_P - (\sigma_P^2 + \mu_P^2)) + \frac{z}{K_{int}} \mu_P (1 - \mu_P) \\
= & - \frac{K_{int} - z}{K_{int}} \sigma_P^2 + \mu_P (1 - \mu_P) \\
b_1 = 1, b_2 = 0 \Rightarrow & \frac{K_{int} - z}{K_{int}} \int_{p_1} p_1 (1 - p_1) \mathcal{N}(p_1) dp_1 + \frac{z}{K_{int}} \int_{p_1} p_1 \mathcal{N}(p_1)(1 - \mu_P) dp_1 \\
= & \frac{K_{int} - z}{K_{int}} (\mu_P - (\sigma_P^2 + \mu_P^2)) + \frac{z}{K_{int}} \mu_P (1 - \mu_P) \\
= & - \frac{K_{int} - z}{K_{int}} \sigma_P^2 + \mu_P (1 - \mu_P) \\
b_1 = 1, b_2 = 1 \Rightarrow & \frac{K_{int} - z}{K_{int}} \int_{p_1} p_1^2 \mathcal{N}(p_1) dp_1 + \frac{z}{K_{int}} \int_{p_1} p_1 \mathcal{N}(p_1) \mu_P dp_1 \\
= & \frac{K_{int} - z}{K_{int}} (\sigma_P^2 + \mu_P^2) + \frac{z}{K_{int}} \mu_P^2 = \frac{K_{int} - z}{K_{int}} \sigma_P^2 + \mu_P^2
\end{aligned}
\tag{8.34}
$$

When adding all the cases for $p_{val}(b_1, b_2)$, the obtained value is 1, confirming that the distribution was calculated correctly. One can finally proceed to the calculation

of the correlation:

$$
\begin{aligned}
&E((b_1-\mu_p)(b_2-\mu_p)) \qquad (8.35)\\
&=\mu_p^2\left(\frac{K_{int}-z}{K_{int}}\sigma_p^2+(\mu_p-1)^2\right)-\mu_p(1-\mu_p)\left(-\frac{K_{int}-z}{K_{int}}\sigma_p^2+\mu_p(1-\mu_p)\right)\\
&-\mu_p(1-\mu_p)\left(-\frac{K_{int}-z}{K_{int}}\sigma_p^2+\mu_p(1-\mu_p)\right)+(1-\mu_p)^2\left(\frac{K_{int}-z}{K_{int}}\sigma_p^2+\mu_p^2\right)\\
&=\frac{K_{int}-z}{K_{int}}\sigma_p^2=\begin{cases}\frac{K_{int}-|z|}{K_{int}}\sigma_p^2 & \text{if } |z|<K_{int}\\ 0 & \text{if } |z|\geq K_{int}\end{cases}
\end{aligned}
$$

The result in Equation 8.35 makes sense because if $K_{int} = 1$ and by setting $z = 1$, it means that a new $p(n)$ value is drawn with every bit sample, and then the correlation goes to zero. As expected correlation goes to zero if $z = K_{int}$ because for sure the two samples will have been generated using two different $p(n)$ drawn from the distribution of $P(n)$. Additionally, if the distribution $P(n)$ didn't have variance, meaning that the output of the random number generators encodes always a fixed probability μ_P, and therefore $\sigma_P^2 = 0$, then correlation between any random bits will also be zero. If one now considered the case depicted in Figure 8.6, where the random number outputs rng are obtained by cycling through all the outputs from the N RNGs, only bits distanced by a multiple of N samples will have correlation, so the correlation in 8.35 can be multiplied by a train of deltas spaced by N samples, and

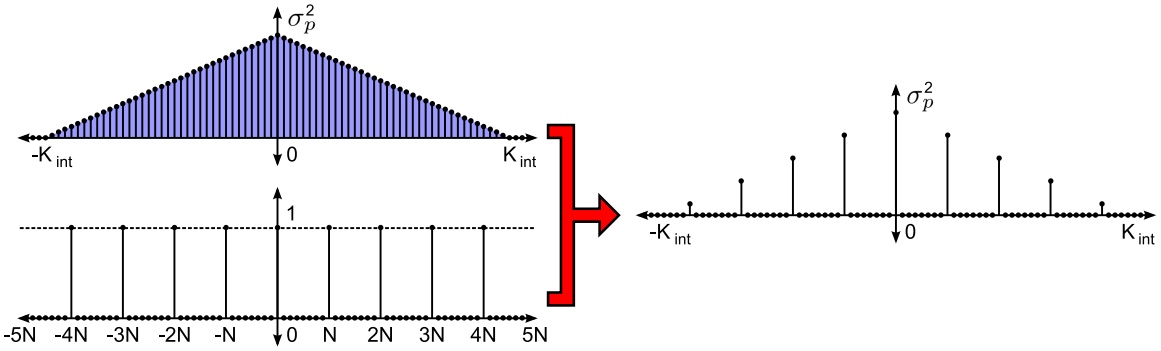


Figure 8.7: Auto-correlation. On the top left corner the auto-correlation of the random number stream from Figure 8.3. On the bottom left corner the train of deltas spaced by N is presented. It is the multiplication of both functions on the left that gives rise to the auto-correlation suffered by any of the scrambled streams in Figure 8.6.

then:

$$E((b_1 - \mu_p)(b_2 - \mu_p)) = \begin{cases} \frac{K_{int}-|z|}{K_{int}}\sigma_p^2 & \text{if } (|z| < K_{int}) \cap (|z| = kN), k \in \mathbb{Z} \\ 0 & \text{otherwise} \end{cases} \tag{8.36}$$

Equation 8.36 has been plotted in Figure 8.7. On the left, the correlation of one of the RNG blocks on top of the train of deltas spaced by N time slots. On the right the multiplication of both the auto-correlation of a single RNG block and the train of deltas. The resulting plot shows the auto-correlation of one of the random streams resulting in the scrambling process shown in Figure 8.6. If $N \geq K_{int}$, then each of the resulting random streams from Figure 8.6 will present no correlation.

8.3 Analog Sigma-Delta Random Number Generator

8.3.1 Architectural Description

Figure 8.8 presents the architecture for the Sigma-Delta random number generator. Depending on the charge added to the capacitor's node from the current source I_1, and the charge subtracted from it through the current source I_2, the mean value at the output $y(n)$ will change. The size of the *pfet* and *nfet* transistors from these current sources can be tweaked in simulations so that the desired output mean can be achieved, but relying on this would presume the non-existence of process variations during fabrication. One way to reduce these fabrication variations would be to increase the size of these transistors. This would seem to be a good idea, but actually it is not. Randomness at the output of this random number generator will come from random telegraph noise (RTN), and RTN is sensitive to transistor sizes. As these sizes are reduced, the effect of this noise is more evident. It is for this reason that transistor sizes for these current sources will be preferably minimum so that one can exploit the randomness of this noise in generating random numbers. In Figure 8.8 it is assumed that current source I_1 is made out of $N1$ transistors in parallel, and I_2 is made out $N2$ transistors in parallel. From now the charge provided by I_1 will be

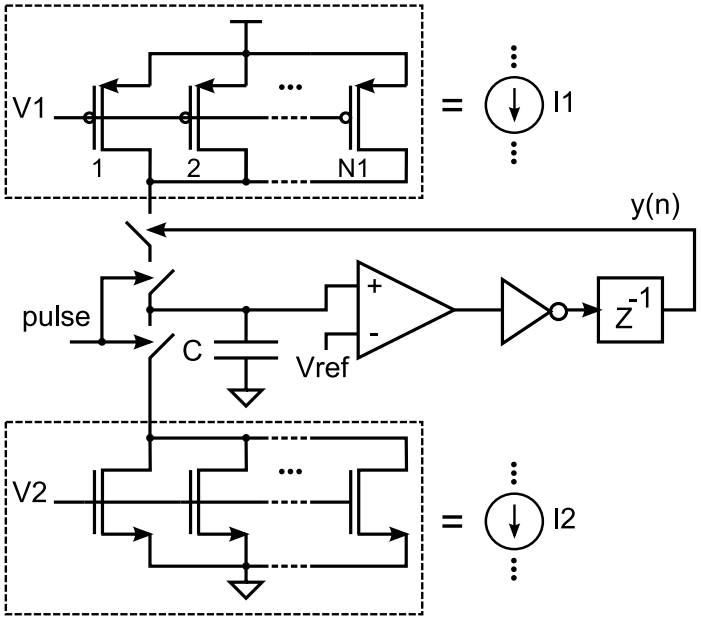


Figure 8.8: Analog Sigma-Delta circuit model.

addressed as Q_1, and the charge withdrawn by I_2 as Q_2.

$$Q_1 = \Delta t\left(\sum_{i=1}^{N1} I_{1i}\right), \text{ and } Q_2 = \Delta t\left(\sum_{i=1}^{N1} I_{2i}\right) \tag{8.37}$$

In Figure 8.9 an equivalent block diagram in the Z domain is presented. In this diagram quantization noise has been again called $QN(z)$, and now $RTN_1(z)$ is the RTN in the I_1 current source transistors and $RTN_2(z)$ is the RTN in the I_2 current source transistors. Additionally, a constant offset is added with ΔV_{ref}. This offset corresponds to the case of a comparator that does not have the threshold voltage set at $VDD/2$, but slightly off.

As one can see in Figure 8.9, all the signals $X(z)$, $QN(z)$, $RTN_1(z)$, $RTN_2(z)$, $Y(z)$ and ΔV_{ref} have been normalized by VDD. One needs to now analyze this LTI

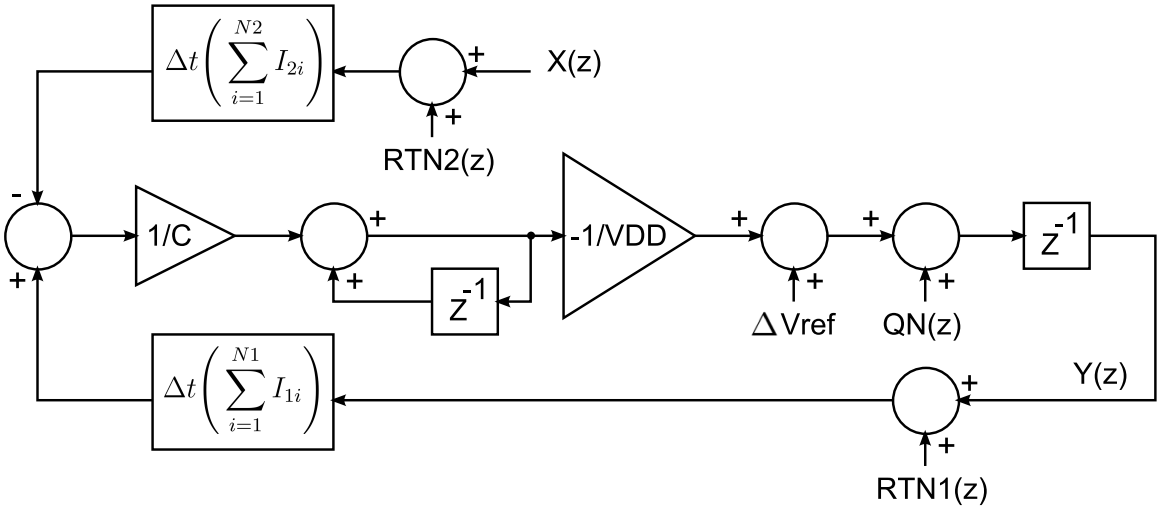


Figure 8.9: Block diagram in the *Z-transform* domain for the Analog Sigma-Delta.

system using superposition for the cases of the different inputs.

1. Case for $X(z)$ and $RTN_2(Z)$.

$$H_1(z) = \frac{Y(z)}{X(z)} = \frac{Y(z)}{RTN_2(z)} = \frac{Q_2}{CVDD}\frac{z^{-1}}{(1 - z^{-1}(1 - \frac{Q_1}{CVDD}))} \tag{8.38}$$

2. Case for $\Delta V_{ref}(z)$ and $QN(z)$.

$$H_2(z) = \frac{Y(z)}{QN(z)} = \frac{z^{-1}(1 - z^{-1})}{(1 - z^{-1}(1 - \frac{Q_1}{CVDD}))} \tag{8.39}$$

3. Case for $RTN_1(z)$.

$$H_3(z) = \frac{Y(z)}{RTN_1(z)} = -\frac{Q_1}{CVDD}\frac{z^{-1}}{(1 - z^{-1}(1 - \frac{Q_1}{CVDD}))} \tag{8.40}$$

The first condition for this system is to be stable. In doing so, the system pole needs to be inside of the unitary circle in the Z domain. The following condition needs to be satisfied:

$$|1 - \frac{Q_1}{VDDC}| < 1 \Rightarrow 2 > \frac{Q_1}{VDDC} > 0 \Rightarrow 2 > \frac{Q_1}{VDDC} \tag{8.41}$$

For the cases of the comparator offset $\Delta V_{ref}(z)$ and $QN(z)$, one can see that their contribution to the mean of $y(n)$ output is none due to the presence of a zero at 1 in their transfer function. Let's now consider $RTN_1 \sim R_1(\mu = \mu_{RTN_1}, \sigma^2_{RTN_1})$, $RTN_2 \sim R_2(\mu = \mu_{RTN_2}, \sigma^2_{RTN_2})$. When the system settles, it then can be said that:

$$\mu_y = E(y(n)) = \frac{Q_2}{Q_1}(1 + \mu_{RTN_2}) + \mu_{RTN_1} \tag{8.42}$$

Random telegraph noise comes from electrons that get trapped at the gate of a transistor, and then they change its V_t. This process can be modeled as a Markov chain, where a maximum number of traps at the gate is N_{traps} and transition matrix P_π can be defined. The noise RTN_1 and RTN_2 can be modeled to be proportional to the two summations of two-states random variables X_{1i} and X_{2i} that can either be 0 or 1:

$$RTN_1 \propto \sum_{i=1}^{N1_{traps}} X_{1i} \text{ and } RTN_2 \propto \sum_{i=1}^{N2_{traps}} X_{2i} \tag{8.43}$$

Auto-correlation $R_{rtn1rtn1}$ and $R_{rtn2rtn2}$ will be a scaled version of the auto-correlation

of the individual X_1 and X_2 variables. Consequently, if one can model the whole system and obtain data from the fabricated chip, parameters such as the number of traps N_{traps}, the probability of getting an electron trapped and the probability of it being released could be estimated. Due to the lack of this type of information at this point, the assumption that the noise inputs RTN_1 and RTN_2 are WSS signals is made.

Auto-correlation R_{yy} is now calculated for the output sequence $y(n)$. $x(n)$ will be a step function, and as a deterministic signal, $R_{xx}(k) = E((X(n)-E(X))(X(n+k)-E(X))) = 0$. The offset $\Delta v_{ref}(n)$ even if it is unknown, after fabrication, it will be a fixed value that will not change over time. This will also make $R_{\Delta v_{ref}\Delta v_{ref}}(k) = 0$. One can then obtain:

$$\begin{aligned} R_{yy} &= h_1(-n) * h_1(n) * R_{rtn_2 rtn_2} + h_2(-n) * h_2(n) * R_{qnqn} \\ &\quad + h_3(-n) * h_3(n) * R_{rtn_1 rtn_1} \end{aligned} \tag{8.44}$$

One can now do:

$$\begin{aligned} &Z(h_1(-n) * h_1(n)) = H_1(z)H_1(z^{-1}) \\ &= -\frac{Q_2^2}{CVDD(CVDD - Q_1)} \frac{z^{-1}}{(1 - z^{-1}(1 - \frac{Q_1}{CVDD}))(1 - z^{-1}(1 - \frac{Q_1}{CVDD})^{-1})} \\ &Z(h_2(-n) * h_2(n)) = H_2(z)H_2(z^{-1}) \\ &= \frac{CVDD}{(CVDD - Q_1)} \frac{(1 - z^{-1})^2}{(1 - z^{-1}(1 - \frac{Q_1}{CVDD}))(1 - z^{-1}(1 - \frac{Q_1}{CVDD})^{-1})} \end{aligned} \tag{8.45}$$

$$= \frac{CVDD}{(CVDD - Q_1)} + \frac{Q_1^2}{(CVDD - Q_1)^2} \frac{z^{-1}}{(1 - z^{-1}(1 - \frac{Q_1}{CVDD}))(1 - z^{-1}(1 - \frac{Q_1}{CVDD})^{-1})} \tag{8.46}$$

$$Z(h_3(-n) * h_3(n)) = H_3(z)H_3(z^{-1})$$
$$= -\frac{Q_1^2}{CVDD(CVDD - Q_1)} \frac{z^{-1}}{(1 - z^{-1}(1 - \frac{Q_1}{CVDD}))(1 - z^{-1}(1 - \frac{Q_1}{CVDD})^{-1})} \tag{8.47}$$

Using the results in Equation 8.18, one can now get:

$$H_1(z)H_1(z^{-1}) = \frac{Q_2^2}{Q_1(2CVDD - Q_1)} \left(\frac{1}{1 - z^{-1}(1 - \frac{Q_1}{CVDD})} - \frac{1}{1 - z^{-1}(1 - \frac{Q_1}{CVDD})^{-1}} \right) \tag{8.48}$$

$$H_2(z)H_2(z^{-1}) = \frac{CVDD}{(CVDD - Q_1)}$$
$$- \frac{Q_1 CVDD}{(CVDD - Q_1)(2CVDD - Q_1)} \left(\frac{1}{1 - z^{-1}(1 - \frac{Q_1}{CVDD})} - \frac{1}{1 - z^{-1}(1 - \frac{Q_1}{CVDD})^{-1}} \right) \tag{8.49}$$

$$H_3(z)H_3(z^{-1}) = \frac{Q_1}{(2CVDD - Q_1)} \left(\frac{1}{1 - z^{-1}(1 - \frac{Q_1}{CVDD})} - \frac{1}{1 - z^{-1}(1 - \frac{Q_1}{CVDD})^{-1}} \right) \tag{8.50}$$

The impulse response for the before mentioned auto-correlation transfer functions

can now be obtained:

$$h_1(-n) * h_1(n) = \frac{Q_2^2}{Q_1(2CVDD - Q_1)}\left(u(n)\left(1 - \frac{Q_1}{CVDD}\right)^n + u(-n+1)\left(1 - \frac{Q_1}{CVDD}\right)^{-n}\right) \tag{8.51}$$

$$h_2(-n) * h_2(n) = \delta(n)\frac{CVDD}{(CVDD - Q_1)} - \frac{Q_1 CVDD}{(CVDD - Q_1)(2CVDD - Q_1)}\left(u(n)\left(1 - \frac{Q_1}{CVDD}\right)^n + u(-n+1)\left(1 - \frac{Q_1}{CVDD}\right)^{-n}\right) \tag{8.52}$$

$$h_3(-n) * h_3(n) = \frac{Q_1}{(2CVDD - Q_1)}\left(u(n)\left(1 - \frac{Q_1}{CVDD}\right)^n + u(-n+1)\left(1 - \frac{Q_1}{CVDD}\right)^{-n}\right) \tag{8.53}$$

Expression for the auto-correlation $R_{yy}(n)$ is now obtained, assuming the noise sources are WSS processes.

$$\begin{aligned} R_{yy}(n) = {} & \frac{CVDD}{(CVDD - Q_1)}\delta(n)\sigma_{qn}^2 \\ & + \frac{(CVDD - Q_1)Q_2^2\sigma_{rtn2}^2 - Q_1^2 CVDD\sigma_{qn}^2 + Q_1^2(CVDD - Q_1)\sigma_{rtn1}^2}{Q_1(2CVDD - Q_1)(CVDD - Q_1)} \\ & \left(u(n)\left(1 - \frac{Q_1}{CVDD}\right)^n + u(-n+1)\left(1 - \frac{Q_1}{CVDD}\right)^{-n}\right) \end{aligned} \tag{8.54}$$

Let's now replace $Q_1 = CVDDX_1$ and $Q_2 = CVDDX_2$, where now X_1 and X_2

are the normalized version to $CVDD$ for Q_1 and Q_2.

$$R_{yy}(n) = \frac{1}{(1-X_1)}\delta(n)\sigma_{qn}^2 + \frac{(1-X_1)X_2^2\sigma_{rtn2}^2 - X_1^2\sigma_{qn}^2 + X_1^2(1-X_1)\sigma_{rtn1}^2}{X_1(2-X_1)(1-X_1)}$$
$$(u(n)(1-X_1)^n + u(-n+1)(1-X_1)^{-n}) \tag{8.55}$$

Let's look at Equation 8.42. The mean μ_y cannot be controlled very well if both the variance in Q_1 and Q_2 due to fabrication are both high. There is a linear dependency of μ_y on Q_2, but the dependency is inversely proportional for the case of Q_1. One would want the variance of Q_1 to be low, so that a better control on μ_y can be applied. Variance of Q_1 low means big transistors, and then this means that no RTN noise will be present for the current source I_1 in Figure 8.9. One can then conclude that adding RTN noise to the feedback transistors is not a good idea. Doing $\sigma_{rtn1}^2 = 0$ one obtains:

$$R_{yy}(n) = \frac{X_1((2-X_1)\delta(n) - X_1)\sigma_{qn}^2 + (1-X_1)X_2^2\sigma_{rtn2}^2}{X_1(2-X_1)(1-X_1)}(u(n)(1-X_1)^n + u(-n+1)(1-X_1)^{-n}) \tag{8.56}$$

From this last expression, the variance can be obtained evaluating $n = 0$.

$$\sigma_y^2 = \frac{X_2^2}{X_1(2-X_1)}\sigma_{rtn2}^2 + \frac{2}{(2-X_1)}\sigma_{QN}^2 \tag{8.57}$$

If a long stream of bits coming from the chip could be recorded, then all the parameters X_1, X_2, σ_{qn}^2 and σ_{rtn2}^2 in $R_{yy}(n)$ could be estimated.

8.3.2 Circuit Description

A description of the analog Sigma-Delta architecture was presented in Subsection 8.2.1, but a schematic for it was not shown. The schematics for the different components in the analog Sigma-Delta are presented here, along with the reasoning behind the choice of the different transistor sizes. From Figure 8.8, the feedback will provide Q_1 of charge in Δt time, and the RTN current source will withdraw Q_2 of charge from the integrating node. In the previous subsection the current provided by the DAC was not mentioned, since it can be considered to be part of Q_2 as an offset. A choice of three bits was made for the DAC, where these three bits could represent signed numbers. A DAC that could inject and withdraw current was then designed where a positive number would inject current into the integrating node, and a negative number would withdraw current. Numbers from -4 to 3 can then be the input to the DAC, and then three selectable *pfet* current sources and four selectable *nfet* current sources needed to be used. An additional selectable *pfet* current source is needed for the feedback, and another selectable *nfet* current source is needed for the RTN noise. The different components involved in the injection of current into the integrating node are shown in Figure 8.10. Figure 8.11 shows the general schematic for the analog Sigma-Delta.

The blocks presented in Figure 8.10 with the pin *c_node_io*, connect to the integrating node as seen in Figure 8.11, and the other blocks are the ones to which a bias current will be provided externally. These biases are provided through the

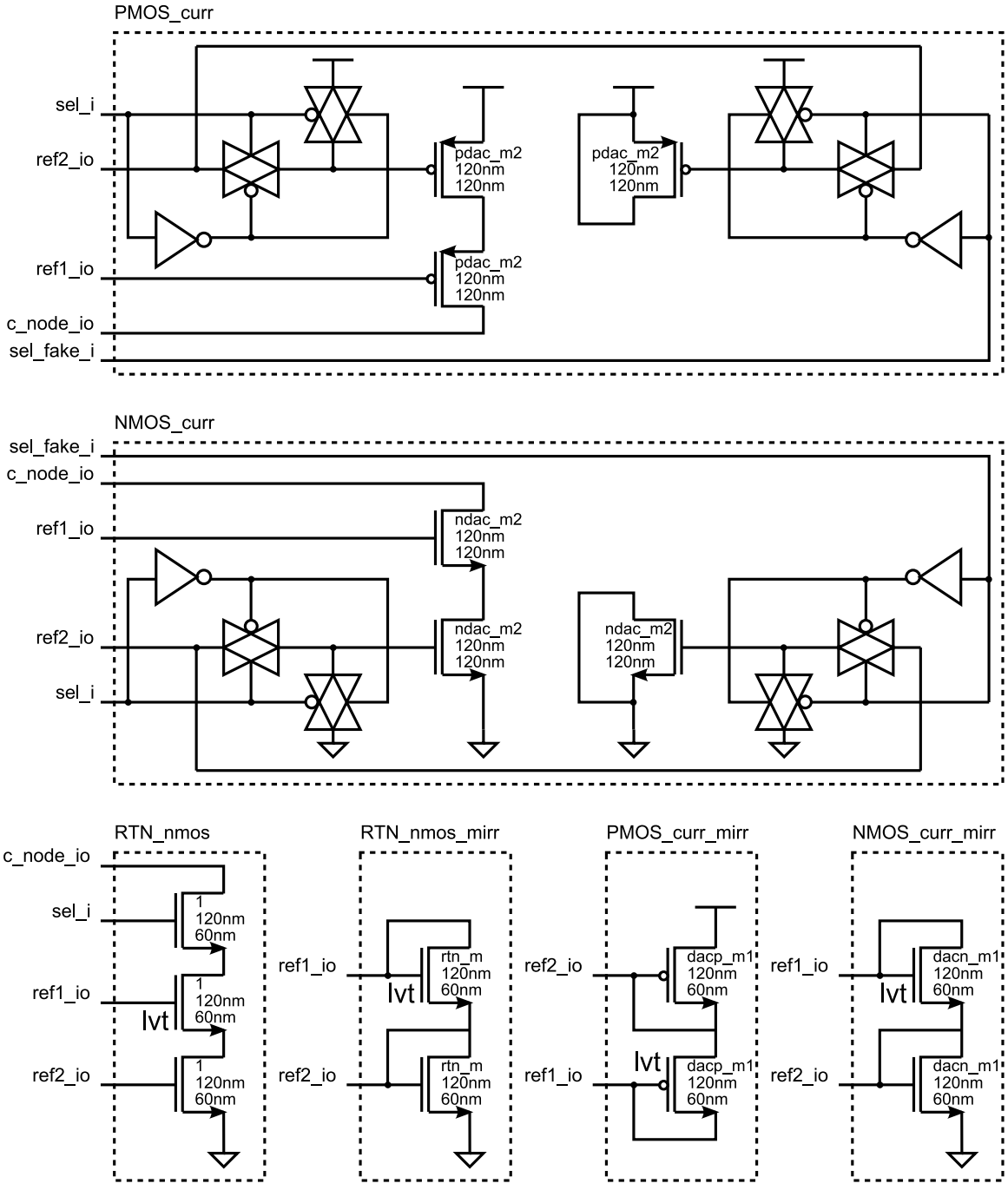


Figure 8.10: Components used in the Sigma-Delta based RNG.

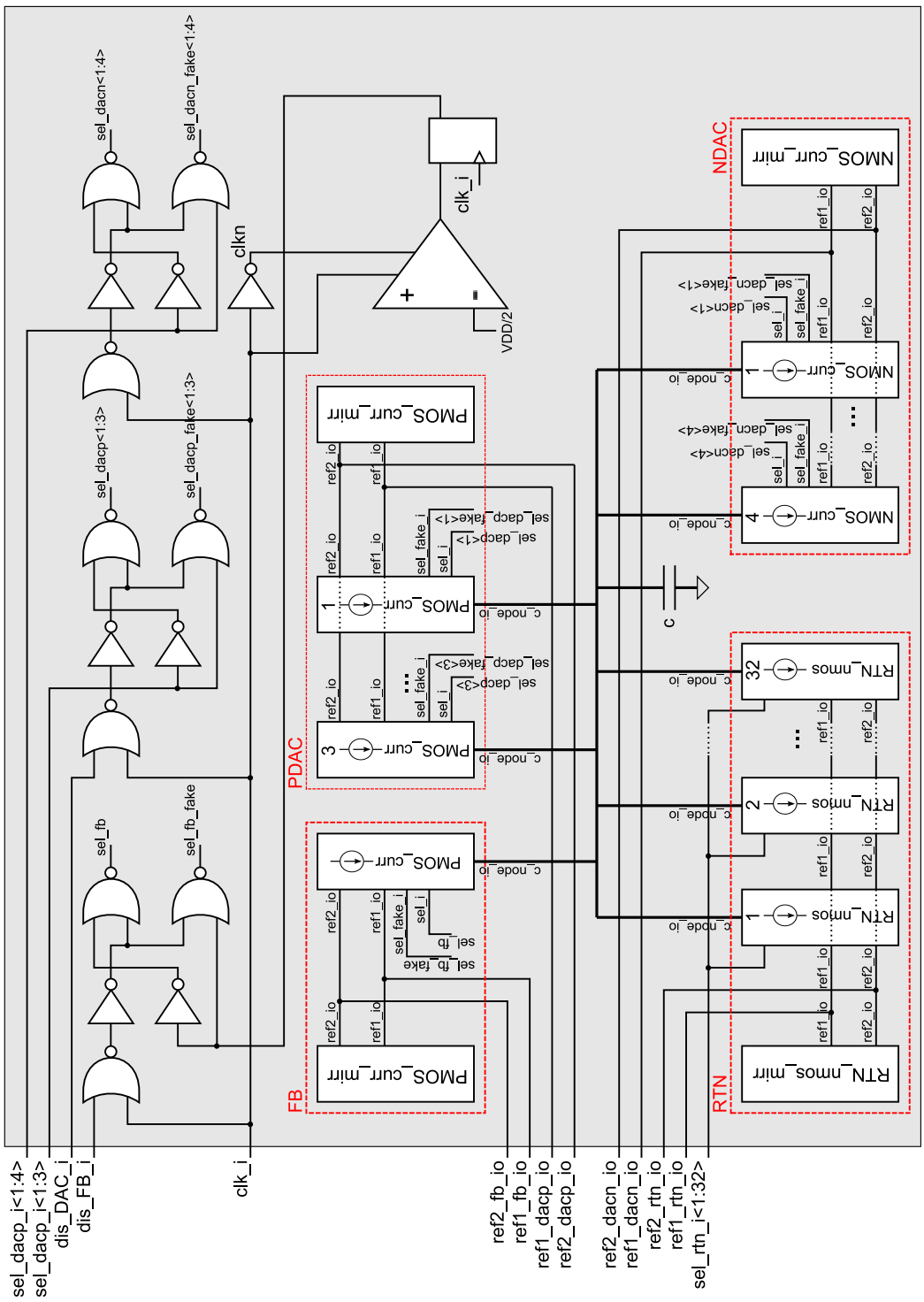


Figure 8.11: General structure for the analog Sigma-Delta based RNG. The circuits for some of the different components can be seen in Figure 8.10.

ref1_io inputs in Figure 8.10. In Figure 8.11 the four biases corresponding to the RTN transistor current, feedback current, and the two biases for the positive and negative numbers in the DAC are provided through the pins *ref1_rtn_io*, *ref1_fb_io*, *ref1_dacn_io* and *ref1_dacp_io*. The choice of cascoded current sources was done considering that the effect of channel modulation was really pronounced. For operating conditions in which the integrating node is kept between $0.3V$ and $0.9V$, and currents are kept below $250nA$, currents would not change more than 5% for the case of cascoded current sources, but up to a 40% change in current was found when using simple current mirrors. When running MonteCarlo on the mirrored current for the RTN transistor, currents were found to be distributed as Gamma, and the standard deviation would go from $1.8nA$ to $129nA$ for when a current from $1nA$ to $250nA$ was being mirrored. The distribution for the current can be seen in Figure 8.12. This shows how little control one has over the current mirrored in the transistor that will provide RTN noise. This is the reason why 32 RTN transistors were decided to be placed in each RNG unit from which one can choose, so that the probability of finding a RTN transistor with a better current matching is higher. This is why a select input is found in the block *RTN_nmos*. This block is replicated 32 times in Figure 8.11, and the user will be given the possibility of choosing one among them. Once the best RTN transistor choice is taken, there is no reason for changing the usage of that transistor for generating random numbers, and then this is the reason the control over the current in *RTN_nmos* is done with just a simple control transistor placed in series

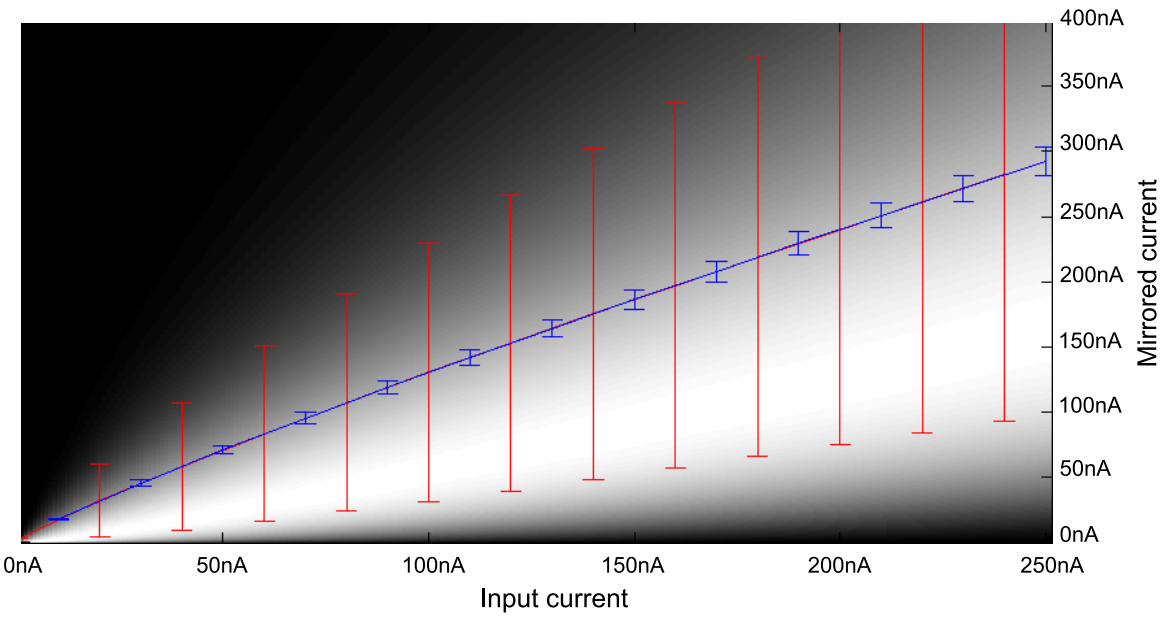


Figure 8.12: Distribution for the mirrored current in the RTN transistor structure. In red the standard deviation and the mean of this current. After applying the choice of one in 32 available RTN transistors, the distribution of currents becomes Gaussian and in blue the new standard deviation is presented, which is observed to be considerable less.

with the cascoded structure. Figure 8.12 shows in red the mean and standard deviation of the current through an RTN transistor, and in blue the standard deviation and mean after applying the choice of one in 32 transistors. It can be observed that the control over the current a RTN transistor provides can be improved dramatically without having to increase the size of the transistors, which would make the RTN noise inherent in small transistors disappear.

For the feedback and DAC current sources, a different approach was taken. In the case of the RTN current sources minimum size transistors were chosen, but for the case of the other current sources, it was found that a better current matching with less variance was achieved when, while maintaining the transistor area, the single

transistor used was square, with dimensions $L = 120nm$ and $W = 120nm$. For the case of these current sources it was not affordable replicating structures and choosing among the ones that work better. Transistor sizing needed to be used to accomplish certain control over the provided current. For applying control over the current, a simple transistor in series with the cascoded current mirror was not a good choice because these current sources are pulsed. If a transistor in series was used, a high amount of charge would build up in the internal nodes of the cascoded structure and the current provided by this current source, for when finally current is allowed to flow, is much more than the desired one. By controlling the gate voltage of the current mirror transistors, this effect is avoided, but unfortunately more power is burnt since the gates of these transistors need to be charged and discharged quite often. This can be seen in both *PMOS_curr* and *NMOS_curr* blocks from Figure 8.10.

For all of the MonteCarlo simulations run for all the currents, the number of transistors in parallel for the transistors in blocks *RTN_nmos_mirr*, *PMOS_curr_mirr* and *NMOS_curr_mirr* was considered to be at least $rtn\_m = 1024$, $dacp\_m1 = 4096$ and $dacn\_m1 = 4096$. This seems to be a pretty large number of transistors in parallel for just one RNG, but if one observes in Figure 8.11, both *ref1* and *ref2* nodes for all of the current sources are pins, meaning that one can reach these high numbers by just sharing the transistors of the before mentioned blocks among all of the RNGs that will be fabricated in a chip. Just as an example, if one decided to fabricate 64 of these controlled RNGs, then, for a single RNG $rtn\_m = 1024/64 = 16$,

*dacp_m*1 = 4096/64 = 64 and *dacn_m*1 = 4096/64 = 64, which is definitely more reasonable. This allows to explain an additional feature added to the *PMOS_curr* and *NMOS_curr* blocks. There is not only one select input, there are actually two *sel_i* and *sel_fake_i*. If *sel_i* is '1', then *sel_fake_i* will be '0' and vice versa. The way of selecting a current source by controlling the gate of the mirroring transistor makes usage of the bias current to charge the gate of that mirroring transistor. The problem is that since the selection of the feedback or DAC current sources is random, this charging is done randomly, making the two voltages *ref1* and *ref2* not necessarily steady over time. If on the other hand one could select every single current source every time, then the mentioned voltages would be steady. This is the reason the fake select input is introduced. If a *PMOS_curr* or *NMOS_curr* block was not selected, charge would still be drawn to charge the *"fake"* current source, making the voltages *ref1_fb_io*, *ref2_fb_io*, *ref1_dacp_io*, *ref2_dacp_io*, *ref1_dacn_io* and *ref2_dacn_io* in Figure 8.11 steady over time.

In Figure 8.13 different values for the parameters *pdac_m2* and *ndac_m2* were consider for *NMOS_curr* and *PMOS_curr* blocks. These figures show the probability distribution of the mirrored currents for both the before-mentioned blocks. These distributions were obtained fitting data from MonteCarlo simulations into Gaussian and gamma distributions in steady state. Bigger numbers for *pdac_m2* and *ndac_m2* are accompanied by a lower variance in the mirrored current, but it also comes with more current leakage through the substrate. This leakage current is additionally

showed in the third column in Figure 8.13. One can observe that the *nfet* transistors, due to the fact that triple well is not being used, present considerable more leakage than the *pfet* transistors. For all of these figures, the leakage current has been removed from the first and second column distributions. With these distributions, a more informed decision can be taken when deciding the values for *pdac_m2* and *ndac_m2*. Due to the fact that the bias current is used to charge in every clock cycle internal nodes from blocks *PMOS_curr* and *NMOS_curr* and that the MonteCarlo simulations were done in steady state, bias currents (except for the RTN one) will have to be scaled up.

In the previous subsection a few conditions were set to make sure the RNGs would work. First of all one needs to make sure that the DAC can compensate for any current offset so that a probability of 0.5 can be achieved. In Equation 8.58 this condition is presented, where I_{RTN} is the current through the RTN transistor, I_{leak} is the total leakage current from all the structures in the analog Sigma Delta, I_{DACP} is the maximum current provided by the *pfet* branch of the DAC, I_{DACN} is the maximum current provided by the *nfet* branch of the DAC, and finally I_{FB} is the feedback current. For the case of the feedback and DAC current sources, it was decided to allow them to provide current only for half clock period. Let's give a quick example for why this is important. Assume during 6 clock cycles we need to allow current to flow only in 3 of them. If the three ones were placed in the first three-time slots or if they were placed in slots 2, 4 and 6, a different total amount of charge

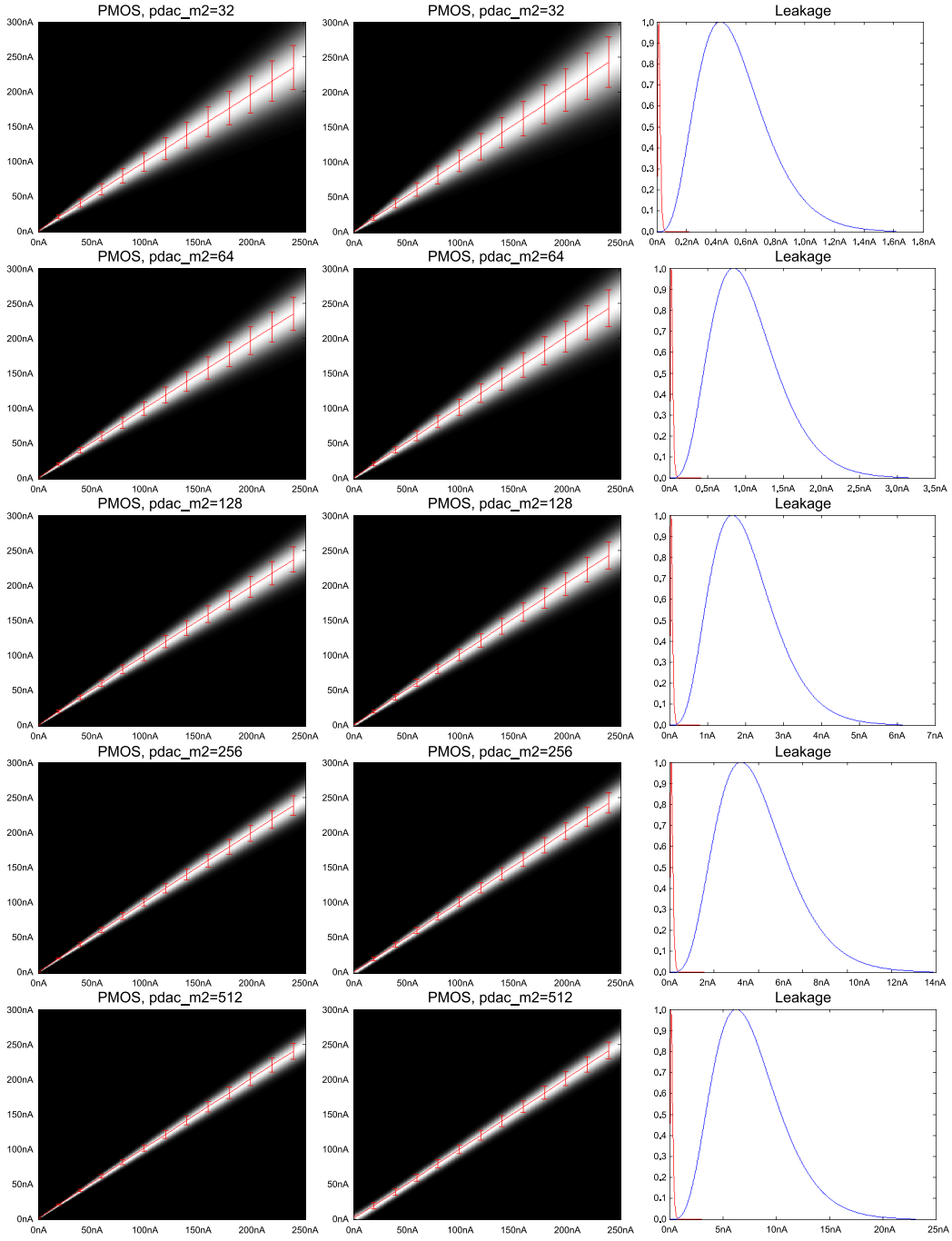


Figure 8.13: Current distributions for *PMOS_curr* and *NMOS_curr* blocks. The first column corresponds to the distribution of current for block *PMOS_curr*, the second column the one for *NMOS_curr*, and finally the third column shows the distribution of leakage current for the *pfet* in red and *nfet* in blue. These distributions are normalized to 1. Every row corresponds to a different value for *pdac_m2* and *ndac_m2*.

would be provided if the high and low transitions for the control signal take different time. This is the reason the DAC and feedback currents in Equation 8.58 are divided by two.

$$\frac{I_{RTN} + I_{leak} - I_{DACP}/2}{I_{FB}/2} < 0.5 < \frac{I_{RTN} + I_{leak} + I_{DACN}/2}{I_{FB}/2} \tag{8.58}$$

Condition in Equation 8.58 is strict, but a more relaxed set of conditions can be set. One needs to make sure that if without applying current from the DAC the probability output is greater than 0.5, then when applying I_{DACP} it is desired to obtain an output probability lower than 0.5. The opposite case needs to be considered for current I_{DACN}. In Equation 8.59 these new set of conditions are presented.

$$\begin{aligned}&\left(\left(\frac{I_{RTN} + I_{leak} - I_{DACP}/2}{I_{FB}/2} < 0.5\right) \text{ AND } \left(\frac{I_{RTN} + I_{leak}}{I_{FB}/2} > 0.5\right)\right) \text{ OR} \\ &\left(\left(\frac{I_{RTN} + I_{leak} + I_{DACN}/2}{I_{FB}/2} > 0.5\right) \text{ AND } \left(\frac{I_{RTN} + I_{leak}}{I_{FB}/2} < 0.5\right)\right)\end{aligned} \tag{8.59}$$

The conditions that make sure the gain AL_0 in Figure 8.3 is less than 1 are now added:

$$\left(\frac{I_{DACP}.4/3}{I_{FB}} < 1\right) \text{ AND } \left(\frac{I_{DACN}}{I_{FB}} < 1\right) \tag{8.60}$$

With the conditions in 8.59 and 8.60 random sampling was applied for when the different bias currents in steps of $5nA$ up to $250nA$ are changed. Considering the different cases for the values of *pdac_m2* and *ndac_m2*, feasible current configurations that would have more than 0.95 probability of satisfying the conditions was found.

A high threshold was set because these conditions need to be satisfied for a RNG to work at all. We found 689, 898, 907, 802 and 619 feasible configurations for the case of 512, 256, 128, 64 and 32 for *pdac_m2* and *ndac_m2*. Based on these numbers we decided to go for $pdac\_m2 = ndac\_m2 = 128$.

Another set of conditions needs to be applied so that one can make sure that the voltage V_{int} at the integrating node remains in a certain range. These conditions are dependent on the frequency of operation F and the capacitance C_{int} in the integrating node. Two situations need to be considered, the cases in which the highest and lowest voltage in V_{int} are achieved. The highest voltage is achieved when at the end of a clock cycle the voltage is close to V_{ref} but is slightly lower, meaning that in the following clock cycle feedback current will be injected. The second case is when one is also close to V_{ref} but slightly above it, meaning that no feedback current will be applied in the following clock cycle. The conditions are now showed in Equation 8.61 and 8.62. Depending on the desired values for the autocorrelation and variance (Equation 8.56 and 8.57) and the probability of satisfying all of the constraints, currents can be

chosen for when the chip is ready to be tested.

$$V_{ref}FC - I_{leak} - I_{RTN} - I_{DACN}/2 > V_{min}FC$$
$$\Rightarrow FC > \frac{I_{leak} + I_{RTN} + I_{DACN}/2}{V_{ref} - V_{min}} \tag{8.61}$$

$$V_{ref}FC - I_{leak} - I_{RTN} + I_{FB}/2 + I_{DACN}/2 < V_{max}FC$$
$$\Rightarrow FC > \frac{-I_{leak} - I_{RTN} + I_{FB}/2 + I_{DACP}/2}{V_{max} - V_{ref}} \tag{8.62}$$

A comparator needed to be designed. A few alternatives were analyzed, but many of them involved sensing the integrating node by extracting some charge from it or involved supplying an additional bias. The sensing method where the sensed value is modified would introduce an additional source of noise, and the whole system would have to be reviewed. Additionally, another bias was not desirable. This is the reason why a bias-less comparator that does not modify the integrating node was designed. Figure 8.14 presents the comparator architecture. For this comparator, feedback had to be used because the voltage at the integrating node is always close to $VDD/2$, and for the two inverter-like structures connected to vin_i, short circuit current would be present for most of the time. It is through Vup and $Vdown$ that feedback was provided. Two similar structures can be seen in Figure 8.14. At the beginning of the comparison (during the time the clock is low) internal nodes $Vup1$, $Vup2$ and $Vup3$ are set high, and nodes $Vdown1$, $Vdown2$ and $Vdown3$ are set low. After the precharge of these nodes, the NAND gate in the circuit will set the node Vup high,

and the NOR gate will set the node $Vdown$ down. This makes the current through the first two inverter-like structure be shut. It is at the transition in which the clock goes low-high that the comparison is made. When this happens, depending on the voltage value in vin_i, if it is lower or higher than the threshold V_{ref} (which should be close to $VDD/2$), either $Vup2$ will transition lower than V_{ref} or $Vdown2$ will transition higher than V_{ref}. When one of these cases happen the inverter chain applied to $Vup2$ and $Vdown2$ will regenerate a digital value and either Vup will go low or $Vdown$ will go high, cutting the short-circuit current. After this, two D flip flops are used to feed the decision taken to the feedback current source through the output out_o, and to send out the random number signal $outd\_o$. The first of the two flip-flops uses the negated clock so that only one clock cycle delay is applied when feeding the feedback decision for the analog Sigma-Delta. *pfet* transistors with their gate connected to VDD and *nfet* transistors with their gate connected to VSS are present in the node precharge logic. The reason for this is just matching purposes when drawing the layout for the comparator. Two transistors can be seen at the top and at the bottom of the comparator schematic. These two have their sources and drains connected, and their sole purpose is to reduce charge injection in the integrating node vin_i. So that a closer voltage to $VDD/2$ can be achieved for V_{ref}, *pfet* and *nfet* transistors were scaled appropriately. MonteCarlo simulations were run for this comparator so that a probability distribution for the V_{ref} could be obtained. In Figure 8.15 the results that were fitted to a Gaussian distribution $\mathcal{N}(\mu = 0.5957V, \sigma = 0.0149)$ are presented.

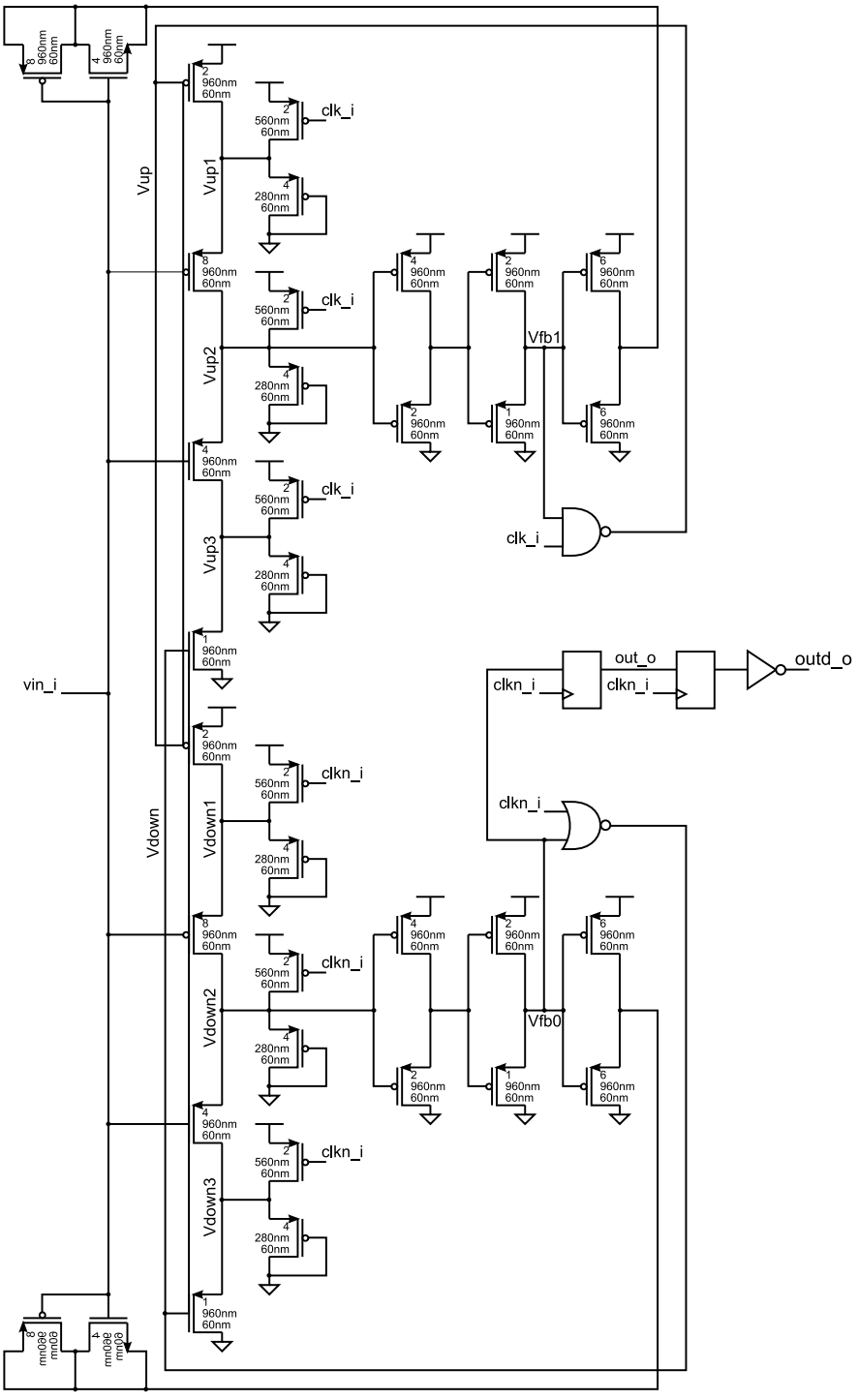


Figure 8.14: Architecture for the comparator used in the analog Sigma-Delta. If the input *vin_i* is lower than the threshold voltage V_{ref} (which is close to $VDD/2$) then a logic one will be sent through *out_o* and a logic zero will be sent through *outd_o*.

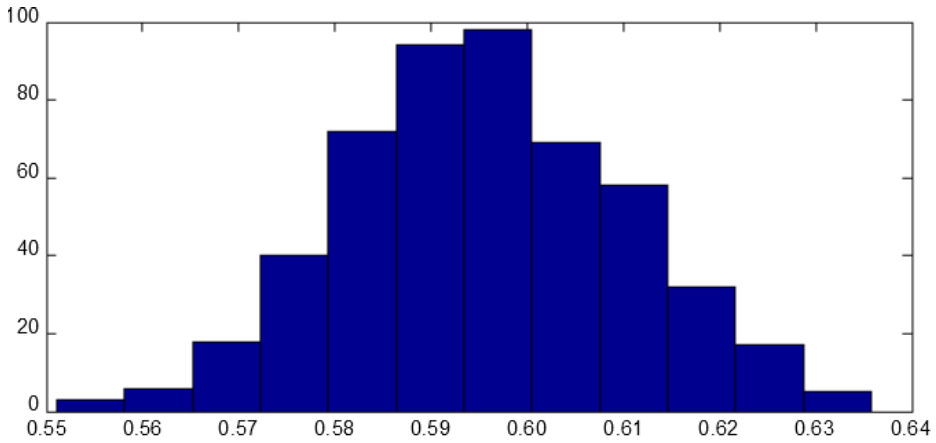


Figure 8.15: Histogram for the comparator threshold voltage V_{ref}. 512 samples were considered. The data was fitted to a Gaussian distribution $\mathcal{N}(\mu = 0.5957V, \sigma = 0.0149)$

For each bit the comparator generates, $169.2fJ$ is used, which is equivalent to using $169.2nW$ @ 1Mhz. After running simulations at different speeds, it was found that the complete analog Sigma-Delta uses $432nW$ (40% corresponds to the comparator) per Mhz.

8.4 TRNG Test Chip GF4

Just like GF2, GF3 and GF5, GF4 was fabricated in *55nm* GF. For the test chip GF4, as we previously mentioned, three bits were chosen for the RNG's DAC. This means that the standard deviation of the probability value encoded at the output of a RNG is restricted to the second column from Tables 8.1, 8.2, 8.3 and 8.4. With the choice of a value for K_{int}, it was finally decided to submit an array of RNG's with different values for K_{int} instead of just one. The eight different K_{int} values from

K_{int}	64	128	256	512	1024	2048	4096	8192
Area (μm^2)	2245.3	2305.4	2367.0	2426.8	2485.4	2543.4	2603.2	2661.8

Table 8.5: Areas for RNG units with different values of K_{int}.

Tables 8.1, 8.2, 8.3 and 8.4 were used for testing purposes. The number of RNGs with the same K_{int} is 18, making the RNG array $18x8 = 144$ units. Eight different designs where synthesized for each of the controlled RNGs displaying the same K_{int}. In Figure 8.16 a block diagram of the whole chip is presented. One can observe that the number of biases supplied to the chip is just four. The other four biases corresponding to the $ref2$ signals are shared among all of the RNGs so that a better current matching can be achieved. The values from Figure 8.10 chosen for the parameters *ndac_m2*, *pdac_m2*, *rtn_m*, *ndac_m1* and *pdac_m1* were 128, 128, 22, 128, 128. For all inputs and outputs, shift registers of $Npipe = 4$ stages were used allowing to achieve higher frequencies of operation. Table 8.5 shows the areas required for RNGs with different K_{int} value. The increase in area due to an increase in the integration time when decoding can be seen is not much.

Table 8.6 present the description of all the input/output in the chip. In Figure 8.17 the layout for the analog Sigma-Delta is displayed. Figure 8.18 shows the layout for the whole chip.

Signal name	Bits	O/I	Description
clk_i	1	I	Clock input.
dis_DAC_i	1	I	This input disables the DAC present in each of the RNG units.
dis_FB_i	1	I	This input disables the feedback in the RNG units.
ref1_fb_io	1	IO	Current bias input for the feedback.
ref1_dacp_io	1	IO	Current bias input for the PMOS DAC.
ref1_dacn_io	1	IO	Current bias input for the NMOS DAC.
ref1_rtn_io	1	IO	Current bias input for the RTN transistors.
rng_sel_i	8	I	This input selects one of the 144 RNGs.
sel_rtn_i	5	I	This 5 bits input addresses one of the 32 RTN transistors present in each RNG unit. With the use of the *rng_sel_i* input a RTN transistor from a particular RNG unit is targeted.
sel_rtn_en_i	1	I	This input stores locally in the targeted RNG unit, the selection of the RTN transistor. This is like a write enable signal.
sd_o	1	O	This is the output of the chip. This output is configured using *rng_sel_i* input that selects from which RNG unit the output is taken.

Table 8.6: Interface signals to the GF4 test chip.

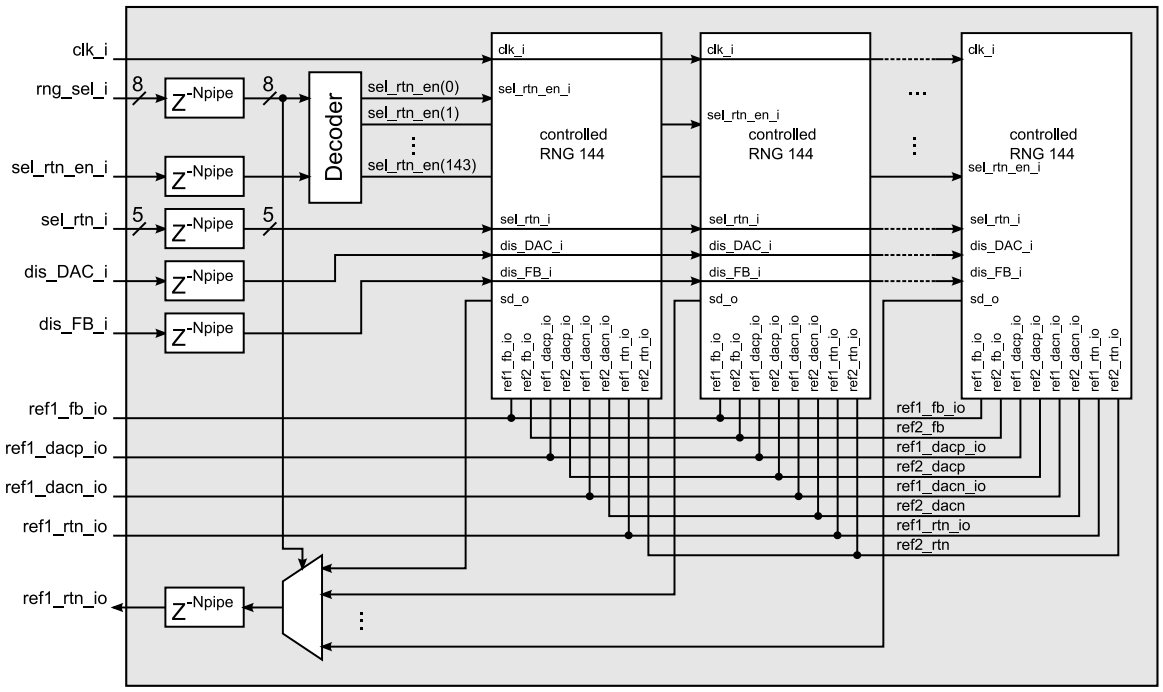


Figure 8.16: Overall diagram of the GF4 test chip.

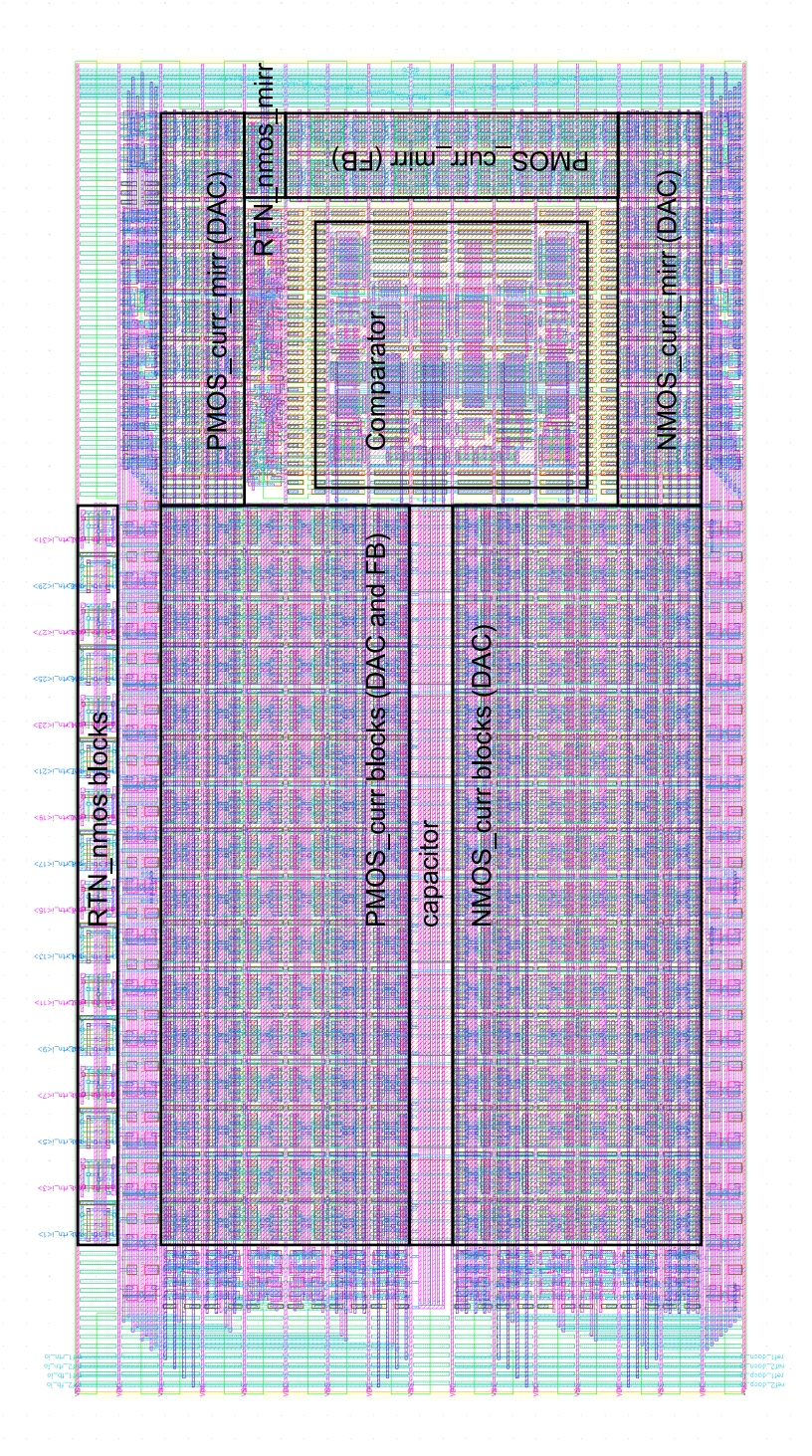


Figure 8.17: Layout for the analog Sigma-Delta based TRNG. Each of the major blocks belonging to this block are showed on the layout.

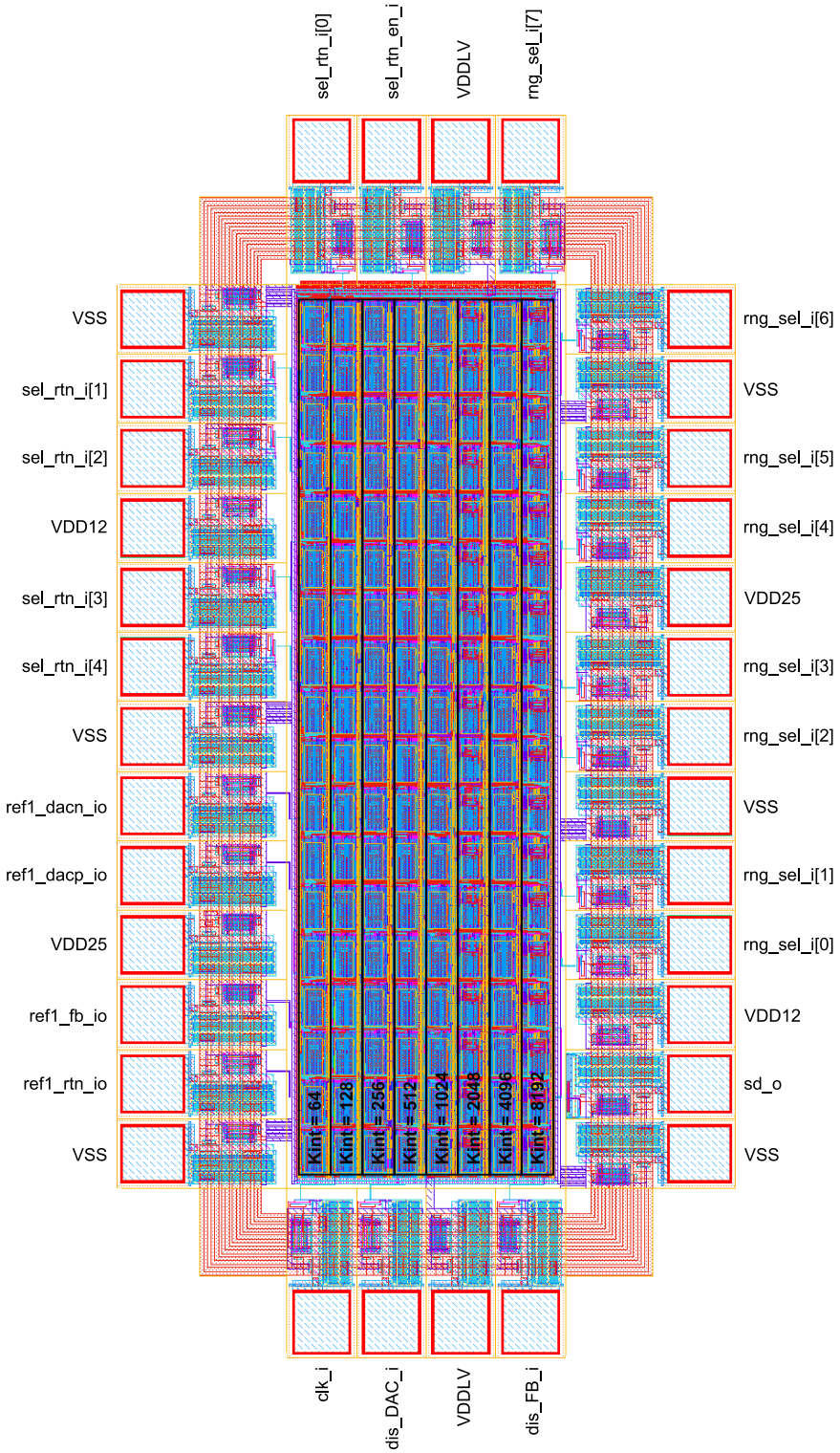


Figure 8.18: GF4 chip layout. The position of each of the pins and the phisical position of the RNG units with the same K_{int} value are presented.

Chapter 9

Conclusions

An energy efficient 2.5D chiplet-based architecture for real-time probabilistic processing of high-velocity sensor data from an autonomous real-time ubiquitous surveillance imaging system has been presented in this dissertation. The work addresses problems at all levels of description.

At the lowest level, not only custom CMOS standard cell libraries were explored and developed, characterizing them at different corners and very different voltage supply values, but also custom cells such as custom SRAM blocks featuring different sizes, asynchronous cells used in the self-diagnosis in the NoCs, monotonically increasing programmable delay lines, double data rate cells, clock tree cells and custom C4/bondpad hybrid pads were designed as well. Furthermore, true random number generator units achieving a Bernoulli probability $p = 0.5$, used in the stochastic encoding of numbers for stochastic processors, were developed exploiting random tele-

graph noise (RTN), intrinsic to single transistor devices. A completely new clock tree distribution network was designed allowing to achieve ultra-low clock skews, ensuring synchronicity over the entire CMP area, which is a characteristic difficult to achieve for chips of this caliber and modularity. An innovative methodology was presented for the design of large scale chips, where the conjunction of modularity and this new clock tree architecture allowed to provide an infrastructure of "half baked" chips, where the designer can radically change the functionality of a chip by modifying the content of the processing units in a matter of minutes. The designer needs to only follow certain guidelines regarding the physical position of power supplies, and the distribution of the pins connecting a processing unit to the mesh of processors.

At the level of the chip architecture, a completely new compact buffer-less switched-circuit mesh network on chip (NoC) capable of reaching very high throughputs ($1.6Tbps$), finite packet delay delivery, free from packet dropping, and free from dead-locks and live-locks was theorized, and finally fabricated for this chiplet-based solution. Additionally, a second NoC connecting processors in the network, was implemented based on token-rings, allowing access to the external DDR memory. Furthermore, a wide bandwidth DRAM physical interface has been designed to address the data flow requirements within and across chiplets. The high speed NoC and physical interface to the DRAM incorporates sub-systems for self-calibration, fault detection, and self-configuration. While no experimental results are available from these sub-systems, extensive simulations using state of the CAD tools show NoC performance in excess

of 300MHz, and pin performance at the physical interface in excess of 1.2 Gbps per each of the 64 bit lines going, and 64 bits coming from the external DDR memory.

At the algorithm and representation levels, the Online Change Point Detection (OCPD) algorithm has been implemented for on-line learning of background-foreground segmentation. Instead of using traditional binary representation of numbers, this architecture relies on unconventional processing of signals using a bio-inspired (spike based) unary representation of numbers. These numbers are represented in a stochastic stream of Bernoulli random variables. By using this representation, probabilistic algorithms can be executed in a native architecture with precision on demand, where if more accuracy is required, more computational time is used. The system architecture has been extensively simulated and validated using state of the art CAD methodology and has been submitted to fabrication in the 55nm CMOS technology. Experimental results from fabricated test chips in the same technology are also presented demonstrating for the system an ASIC implementation of stochastic computation for exact Bayesian inference.

Bibliography

[1] R. Thakkar, "A primer for dissemination services for Wide Aray Motion Imagery," Tech. Rep. OCG 12-077r1, Dec. 2012.

[2] D. J. Brady, M. E. Gehm, R. A. Stack, D. L. Marks, D. S. Kittle, D. R. Golish, E. M. Vera, and S. D. Feller, "Multiscale gigapixel photography," *Nature*, vol. 486, no. 7403, pp. 386–389, Jun. 2012.

[3] D. Brady. (2014, Feb.) AWARE2 Multiscale Gigapixel Camera. [Online]. Available: http://disp.duke.edu/projects/AWARE/

[4] Logos-Technologies, "Multi-Sensor, Wide-Area Persistent Surveillance," pp. 1–2, Sep. 2015.

[5] Harris-Corporation, "CorvusEye 1500," pp. 1–4, 2015.

[6] UTC-AerospaceSystems, "ISR Systems," pp. 1–7, Nov. 2015.

[7] R. Porter, A. Fraser, and D. Hush, "Wide-Area Motion Imagery," *IEEE Signal Processing Magazine*, vol. 27, no. 5, pp. 56–65, Sep. 2010.

[8] Logos-Technologies, "Simera: Lightweight, Wide-Area Persistent Surveillance Sensor for Aerostats," pp. 1–2, Sep. 2015.

[9] E. Culurciello, R. Etienne-Cummings, and K. A. Boahen, "A biomorphic digital image sensor," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 2, pp. 281–294, Feb. 2003.

[10] E. Culurciello and A. G. Andreou, "CMOS image sensors for sensor networks," *Analog Integrated Circuits and Signal Processing*, vol. 49, no. 1, pp. 39–51, Oct. 2006.

[11] P. Lichtsteiner, C. Posch, and T. Delbruck, "A 128x128 120dB 15us Latency Asynchronous Temporal Contrast Vision Sensor," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 2, pp. 566–576, 2008.

[12] C. G. Rizk, P. O. Pouliquen, and A. G. Andreou, "Flexible Readout and Integration Sensor (FRIS): new class of imaging sensor arrays optimized for air and missile defense," *Johns Hopkins APL Technical Digest*, vol. 28, no. 3, pp. 252–253, Jan. 2010.

[13] J. H. Lin, P. O. Pouliquen, A. G. Andreou, A. C. Goldberg, and C. G. Rizk, "Flexible readout and integration sensors (FRIS): a bio-inspired, system-on-chip, event based readout architecture," in *Proceedings of SPIE: Infrared Technology and Applications XXXVIII Conference*, May 2012, pp. 8353–1N.

[14] C. Stauffer and W. E. L. Grimson, "Learning patterns of activity using real-time tracking," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, pp. 747–757, 2000.

[15] SDMS. (2006) Columbus Large Image Format (CLIF-2006) Dataset.

[16] R. T. Collins, X. Zhou, and S. K. Teh, "An open source tracking testbed and evaluation web site," in *IEEE International Workshop on Performance Evaluation of Tracking and Surveillance (PETS 2005)*, 2005.

[17] A. Rajaram and D. Z. Pan, "Robust chip-level clock tree synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 6, pp. 877–890, June 2011.

[18] C. Yeh, G. Wilke, H. Chen, S. Reddy, H. Nguyen, T. Miyoshi, W. Walker, and R. Murgai, "Clock distribution architectures: a comparative study," in *7th International Symposium on Quality Electronic Design (ISQED'06)*, March 2006, pp. 7 pp.–91.

[19] P. Baran, "On distributed communications networks," *IEEE Transactions on Communications Systems*, vol. 12, no. 1, pp. 1–9, March 1964.

[20] U. Feige and P. Raghavan, "Exact analysis of hot-potato routing," in *Proceedings., 33rd Annual Symposium on Foundations of Computer Science*, Oct 1992, pp. 553–562.

[21] T. Moscibroda and O. Mutlu, "A case for bufferless routing in on-chip networks." Association for Computing Machinery, Inc., June 2009. [Online]. Available: https://www.microsoft.com/en-us/research/publication/a-case-for-bufferless-routing-in-on-chip-networks/

[22] C. Fallin, C. Craik, and O. Mutlu, "Chipper: A low-complexity bufferless deflection router," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, Feb 2011, pp. 144–155.

[23] E. J. Mentze, H. L. Hess, K. M. Buck, and D. F. Cox, "Low voltage to high voltage level shifter and related methods," Patent 7 112 995, September, 2006. [Online]. Available: http://www.freepatentsonline.com/7112995.html

[24] K. Chen and K. Chen, "Integrated circuit for level-shifting voltage levels," Dec. 19 2006, uS Patent 7,151,391. [Online]. Available: https://www.google.com/patents/US7151391

[25] K. Joe, H. David, and F. Cox, "F.cox, "level shifting interfaces for low voltage logic," in *9 th NASA Symposium on VLSI Design 2000*, 2000, p. 4.

[26] P. O. Pouliquen, "A ratioless and biasless static cmos level shifter," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, May 2010, pp. 4097–4100.

[27] R. Ginosar, "Metastability and synchronizers: A tutorial," *IEEE Design Test of Computers*, vol. 28, no. 5, pp. 23–35, Sept 2011.

[28] O. Albert and C. F. Mecklenbräuker, "An 8-bit programmable fine delay circuit with step size 65ps for an ultrawideband pulse position modulation testbed," in *2007 15th European Signal Processing Conference*, Sept 2007, pp. 1840–1843.

[29] B. Arkin, "Programmable delay circuit having calibratable delays," Oct. 5 1999, uS Patent 5,963,074. [Online]. Available: https://www.google.com/patents/US5963074

[30] D. Murakami and T. Kuwabara, "A digitally programmable delay line and duty cycle controller with picosecond resolution," in *Proceedings of the 1991 Bipolar Circuits and Technology Meeting*, Sep 1991, pp. 218–221.

[31] B. I. Abdulrazzaq, I. A. Halin, R. M. Sidek, S. Shafie, N. A. M. Yunus, and S. Kawahito, "Sub-picosecond jitter resolution wide range digital delay line for soc integration," in *2015 IEEE International Circuits and Systems Symposium (ICSyS)*, Sept 2015, pp. 44–48.

[32] E. Traa, "Programmable high-speed digital delay circuit," Sep. 12 1989, uS Patent 4,866,314. [Online]. Available: https://www.google.com/patents/US4866314

[33] I. Sourikopoulos, A. Frappé, A. Cathelin, L. Clavier, and A. Kaiser, "A digital

delay line with coarse/fine tuning through gate/body biasing in 28nm fdsoi," in *ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference*, Sept 2016, pp. 145–148.

[34] A. Sagahyroon, J. Placer, M. Burmood, and M. Massoumi, "A vhdl-based simulation methodology for estimating switching activity in static cmos circuits," in *Proceedings Eleventh Annual IEEE International ASIC Conference (Cat. No.98TH8372)*, Sep 1998, pp. 295–300.

[35] J. Juan-Chico, J. Bellido, P. Ruiz-De-Clavijo, C. Baena, C. J. Jimenez, and M. Valencia, "Switching activity evaluation of cmos digital circuits using logic timing simulation," *Electronics Letters*, vol. 37, no. 9, pp. 555–557, Apr 2001.

[36] G. Ascia, V. Catania, M. Palesi, and A. Parlato, "Switching activity reduction in embedded systems: a genetic bus encoding approach," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 6, pp. 756–764, Nov 2005.

[37] N. Lotze and Y. Manoli, "A 62mv 0.13 μ m cmos standard-cell-based design technique using schmitt-trigger logic," in *2011 IEEE International Solid-State Circuits Conference*, Feb 2011, pp. 340–342.

[38] S. Hanson, B. Zhai, M. Seok, B. Cline, K. Zhou, M. Singhal, M. Minuth, J. Olson, L. Nazhandali, T. Austin, D. Sylvester, and D. Blaauw, "Performance and variability optimization strategies in a sub-200mv, 3.5pj/inst, 11nw subthreshold processor," in *2007 IEEE Symposium on VLSI Circuits*, June 2007, pp. 152–153.

[39] B. Zhai, L. Nazhandali, J. Olson, A. Reeves, M. Minuth, R. Helfand, S. Pant, D. Blaauw, and T. Austin, "A 2.60pj/inst subthreshold sensor processor for optimal energy efficiency," in *2006 Symposium on VLSI Circuits, 2006. Digest of Technical Papers.*, June 2006, pp. 154–155.

[40] S. Hanson, B. Zhai, M. Seok, B. Cline, K. Zhou, M. Singhal, M. Minuth, J. Olson, L. Nazhandali, T. Austin, D. Sylvester, and D. Blaauw, "Exploring variability and performance in a sub-200-mv processor," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 4, pp. 881–891, April 2008.

[41] M. Seok, G. Chen, S. Hanson, M. Wieckowski, D. Blaauw, and D. Sylvester, "Cas-fest 2010: Mitigating variability in near-threshold computing," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1, no. 1, pp. 42–49, March 2011.

[42] S. Luetkemeier, T. Jungeblut, M. Porrmann, and U. Rueckert, "A 200mv 32b subthreshold processor with adaptive supply voltage control," in *2012 IEEE International Solid-State Circuits Conference*, Feb 2012, pp. 484–486.

[43] Y. Nakagome, M. Horiguchi, T. Kawahara, and K. Itoh, "Review and future prospects of low-voltage ram circuits," *IBM Journal of Research and Development*, vol. 47, no. 5.6, pp. 525–552, Sept 2003.

[44] S. Lutkemeier, T. Jungeblut, H. K. O. Berge, S. Aunet, M. Porrmann, and U. Ruckert, "A 65 nm 32 b subthreshold processor with 9t multi-vt sram and

adaptive supply voltage control," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 1, pp. 8–19, Jan 2013.

[45] C. Stauffer and W. E. L. Grimson, "Adaptive background mixture models for real-time tracking," in *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*, vol. 2, 1999, p. 252 Vol. 2.

[46] P. KaewTraKulPong and R. Bowden, *An Improved Adaptive Background Mixture Model for Real-time Tracking with Shadow Detection.* Boston, MA: Springer US, 2002, pp. 135–144. [Online]. Available: https://doi.org/10.1007/978-1-4615-0913-4_11

[47] P. Li and D. J. Lilja, "A low power fault-tolerance architecture for the kernel density estimation based image segmentation algorithm," in *ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, Sept 2011, pp. 161–168.

[48] R. P. Adams and D. J. MacKay, "Bayesian Online Changepoint Detection," *arXiv.org*, p. 3742, Oct. 2007.

[49] D. Lunn, C. Jackson, N. Best, A. Thomas, and D. Spiegelhalter, *The BUGS Book: A Practical Introduction to Bayesian Analysis*, 1st ed., ser. Texts in Statistical Science. Chapman & Hall/CRC, Oct. 2012.

[50] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin, *Bayesian Data Analysis*, 2nd ed., ser. Texts in Statistical Science. Chapman & Hall/CRC, Feb. 2006.

[51] A. Smith, "A Bayesian approach to inference about a change-point in a sequence of random variables," *Biometrika*, vol. 62, no. 2, pp. 407–416, 1975.

[52] J. J. O Ruanaidh, W. J. Fitzgerald, and K. J. Pope, "Recursive Bayesian location of a discontinuity in time series," in *Proceedings of the 1994 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 1994, pp. 513–516.

[53] R. Turner, Y. Saatçi, and C. E. Rasmussen, "Adaptive sequential Bayesian change point detection," Tech. Rep., 2009.

[54] J. Mellor and J. Shapiro, "Thompson Sampling in Switching Environments with Bayesian Online Change Detection," in *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2013.

[55] Y. Saatçi, R. D. Turner, and C. E. Rasmussen, "Gaussian process change point models," in *Proceedings of the 27th Annual International Conference on Machine Learning (ICML-10)*, 2010, pp. 927–934.

[56] V. K. Mansinghka, E. M. Jonas, and J. E. Tenenbaum, "Stochastic digital circuits for probabilistic inference," *MIT-CSAIL Technical Report*, vol. TR-2008-069, Nov. 2008.

[57] B. R. Gaines, "Techniques of identification with the stochastic computer," in *1967 IFAC Symposium on The Problems of Identification in Automatic Control Systems*. 1967 IFAC Symposium on the problems of identification in automatic control systems, 1967, pp. 1–18.

[58] A. Alaghi and J. P. Hayes, "Survey of stochastic computing," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, pp. 92:1–92:19, May 2013. [Online]. Available: http://doi.acm.org/10.1145/2465787.2465794

[59] B. D. Brown and H. C. Card, "Stochastic neural computation. i. computational elements," *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 891–905, Sep 2001.

[60] ——, "Stochastic neural computation. ii. soft competitive learning," *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 906–920, Sep 2001.

[61] V. Canals, A. Morro, and J. L. Rossello, "Stochastic based pattern recognition analysis," *Pattern Recognition Letters*, vol. 31, no. 15, pp. 2353 – 2356, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016786551000231X

[62] P. Li and D. J. Lilja, "Using stochastic computing to implement digital image processing algorithms," in *2011 IEEE 29th International Conference on Computer Design (ICCD)*, Oct 2011, pp. 154–161.

[63] P. Li, D. J. Lilja, W. Qian, K. Bazargan, and M. Riedel, "The synthesis of complex arithmetic computation on stochastic bit streams using sequential logic," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '12. New York, NY, USA: ACM, 2012, pp. 480–487. [Online]. Available: http://doi.acm.org/10.1145/2429384.2429483

[64] W. Qian and M. Riedel, "The synthesis of robust polynomial arithmetic with stochastic logic," in *Proceedings of the 45th ACM/EDAC/IEEE Design Automation Conference (DAC'08)*, 2008, pp. 648–653.

[65] A. N. Burkitt, "A review of the integrate-and-fire neuron model: I. homogeneous synaptic input," *Biological Cybernetics*, vol. 95, no. 1, pp. 1–19, Jul 2006. [Online]. Available: https://doi.org/10.1007/s00422-006-0068-6

[66] S. Dreiseitl and L. Ohno-Machado, "Logistic regression and artificial neural network classification models: a methodology review," *Journal of Biomedical Informatics*, vol. 35, no. 5, pp. 352 – 359, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1532046403000340

[67] C. J. Petrie C.S., "A noise-based ic random number generator for applications in cryptography," *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on (Volume:47, Issue:5)*, pp. 612–621, 2000.

[68] T. T. Von Kaenel V., "Dual true random number generators for cryptographic

applications embedded on a 200 million device dual cpu soc," *Custom Integrated Circuits Conference, 2007. CICC '07. IEEE*, pp. 269–272, 2007.

[69] B. D. M. T. Tokunaga C., "True random number generator with a metastability-based quality control," *Solid-State Circuits, IEEE Journal of (Volume:43, Issue:1)*, pp. 78–85, 2008.

[70] B. S. Holleman J., "A 3 μ w cmos true random number generator with adaptive floating-gate offset cancellation," *Solid-State Circuits, IEEE Journal of (Volume:43 , Issue: 5)*, pp. 1324 – 1336, 2008.

[71] S. Srinivasan S., Mathew, "A 4gbps 0.57pj/bit process-voltage-temperature variation tolerant all-digital true random number generator in 45nm cmos," *VLSI Design, 2009 22nd International Conference on*, pp. 301–306, 2009.

[72] M. S. Srinivasan S., "2.4ghz 7mw all-digital pvt-variation tolerant true random number generator in 45nm cmos," *VLSI Circuits (VLSIC), 2010 IEEE Symposium on*, pp. 203–204, 2010.

[73] T. Figliolia, P. Julian, G. Tognetti, and A. G. Andreou, "A true random number generator using rtn noise and a sigma delta converter," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2016, pp. 17–20.

[74] K. K. Hung, P. K. Ko, C. Hu, and Y. C. Cheng, "Random telegraph noise of

deep-submicrometer mosfets," *IEEE Electron Device Letters*, vol. 11, no. 2, pp. 90–92, Feb 1990.

[75] A. K. M. M. Islam and H. Onodera, "Effect of supply voltage on random telegraph noise of transistors under switching condition," in *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, Sept 2017, pp. 1–8.

[76] E. Abbaspour, S. Menzel, and C. Jungemann, "Random telegraph noise analysis in redox-based resistive switching devices using kmc simulations," in *2017 International Conference on Simulation of Semiconductor Processes and Devices (SISPAD)*, Sept 2017, pp. 313–316.

[77] F. M. Puglisi, A. Padovani, L. Larcher, and P. Pavan, "Random telegraph noise: Measurement, data analysis, and interpretation," in *2017 IEEE 24th International Symposium on the Physical and Failure Analysis of Integrated Circuits (IPFA)*, July 2017, pp. 1–9.

[78] Z. Lin, S. Guo, R. Wang, D. Mao, and R. Huang, "A simple method to identify metastable states in random telegraph noise (rtn)," in *2017 IEEE 24th International Symposium on the Physical and Failure Analysis of Integrated Circuits (IPFA)*, July 2017, pp. 1–4.

[79] C. Rizk, F. Tejada, J. Hughes, D. Barbehenn, P. Pouliquen, and A. G. Andreou, "Characterization of rtn noise in the analog front-end of digital pixel imagers,"

in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2017, pp. 1–4.

[80] J. M. de la Rosa, "Sigma-Delta Modulators: Tutorial Overview, Design Guide, and State-of-the-Art Survey," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 1, pp. 1–21, Jan. 2011.

[81] D. Jarman, "A brief introduction to sigma delta conversion," Intersil, Tech. Rep., 1995.

[82] R. Schreier and G. C. Temes, *Understanding delta-sigma data converters*. New York, NY: Wiley, 2005. [Online]. Available: https://cds.cern.ch/record/733538

[83] P. Julian, A. Desages, and B. D'Amico, "Orthonormal high-level canonical pwl functions with applications to model reduction," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 47, no. 5, pp. 702–712, May 2000.

[84] P. Julian, R. Dogaru, and L. O. Chua, "A piecewise-linear simplicial coupling cell for cnn gray-level image processing," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 49, no. 7, pp. 904–913, Jul 2002.

[85] S. J. Thorpe, A. Delorme, and R. Van Rullen, "Spike-based strategies for rapid processing," *Neural Networks*, vol. 14, no. 6-7, pp. 715–725, 2001.

[86] J. L. Molin, A. Eisape, C. S. Thakur, V. Varghese, C. Brandli, and R. Etienne-Cummings, "Low-power, low-mismatch, highly-dense array of vlsi mihalas-niebur neurons," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2017, pp. 1–4.

[87] R. Karakiewicz, R. Genov, and G. Cauwenberghs, "1.1 tmacs/mw fine-grained stochastic resonant charge-recycling array processor," *IEEE Sensors Journal*, vol. 12, no. 4, pp. 785–792, April 2012.

[88] S. Imai, "Cepstral analysis synthesis on the mel frequency scale," in *ICASSP '83. IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 8, Apr 1983, pp. 93–96.

[89] A. Tomar, R. K. Pokharel, O. Nizhnik, H. Kanaya, and K. Yoshida, "Design of 1.1 ghz highly linear digitally-controlled ring oscillator with wide tuning range," in *2007 IEEE International Workshop on Radio-Frequency Integration Technology*, Dec 2007, pp. 82–85.

[90] C. M. Andreou, S. Koudounas, and J. Georgiou, "A novel wide-temperature-range, 3.9 ppm c cmos bandgap reference circuit," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 2, pp. 574–581, 2012.

[91] T. Delbruck and P. Lichtsteiner, "Fully programmable bias current generator with 24 bit resolution per bias," in *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on.* IEEE, 2006, pp. 4–pp.

Vita

Tomás Figliolia received his Engineer's degree with honors in Electronic Engineering from the University of Buenos Aires in 2009, and enrolled in the Electrical and Computer Engineering Ph.D. program at Johns Hopkins University in 2010. That fall he began his graduate studies under the mentorship of Dr. Andreas G. Andreou who introduced him to the neuromorphic engineering field. Tomás Figliolia earned his M.S. in Electrical and Computer Engineering from Johns Hopkins University in 2011. With Dr. Andreou's guidance Tomás has worked from machine learning, signal processing and statistics, to device physics and the fabrication of several microchips. The biggest challenge in Tomás' Ph.D. program was the design of four 17.466mm by 14.133mm chips supporting multi-NoC multi-processor interconnection with high-speed connection to external 3D DDR memory, for which a $55nm$ Global Foundries wafer run was used.

VITA